HZ BOOKS
华章IT

[PACKT]
PUBLISHING

深入探讨Flume重要组件及在日常开发中的使用与最佳实践，涵盖构建高可用、容错、流式数据管道所需的一切知识

以案例驱动方式详细讲解MapReduce模式，深入浅出地介绍Hadoop及MapReduce编程知识

大数据
技术丛书

Apache Flume: Distributed Log Collection for Hadoop
Instant MapReduce Patterns-Hadoop Essentials How-to

Flume日志收集与MapReduce模式

[美] 史蒂夫·霍夫曼（Steve Hoffman）
斯里纳特·佩雷拉（Srinath Perera） 著

张龙 译

机械工业出版社
China Machine Press

内容简介

本书分为上下两篇，对Flume重要组件及其在日常开发中的使用以及MapReduce编程知识进行了全面而深入的探讨，提供大量实践案例，可以帮助读者快速掌握并灵活运用Flume和MapReduce知识解决实际项目中遇到的问题。

上篇介绍Flume的重要组件以及如何用Flume解决HDFS和流式数据/日志的问题，首先简要介绍Flume的架构，包括将数据移动到数据库以及从数据库获取数据、NoSQL数据存储和性能调优，然后深入讲解各个架构组件（源、通道、接收器、通道处理器、接收器组等）的具体实现及配置选项，并且介绍了如何编写自定义的实现，最后介绍Flume监控方面的知识并总结了实时分布式数据收集的现状。

下篇则对Hadoop以及MapReduce编程进行了简明介绍，旨在帮助读者快速起步并对使用Hadoop进行编程有个总体的认识。本篇主要内容包括如何编写一个Hadoop数据格式化器来读取Amazon数据格式，如何通过MapReduce处理Amazon数据、连接两个数据集、实现差集、统计两个条目同时出现的次数、实现图的遍历，以及如何通过反向索引实现简单的搜索，如何通过Kmeans算法建立数据集的集群等。

大数据技术丛书

Flume日志收集与MapReduce模式

[美] 史蒂夫·霍夫曼（Steve Hoffman）
斯里纳特·佩雷拉（Srinath Perera） 著

张龙 译

Apache Flume: Distributed Log Collection for Hadoop
Instant MapReduce Patterns-Hadoop Essentials How-to

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Flume 日志收集与 MapReduce 模式 /（美）霍夫曼（Hoffman, S.），（美）佩雷拉（Perera, S.）著；张龙译 .—北京：机械工业出版社，2015.5（2015.11 重印）
（大数据技术丛书）
书名原文：Apache Flume: Distributed Log Collection for Hadoop Instant MapReduce Patterns-Hadoop Essentials How-to

ISBN 978-7-111-50207-4

I. F… II. ①霍… ②佩… ③张… III. 数据采集 IV. TP274

中国版本图书馆 CIP 数据核字（2015）第 098968 号

Flume 日志收集与 MapReduce 模式

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：陈佳媛
责任校对：董纪丽
印　　刷：三河市宏图印务有限公司
版　　次：2015 年 11 月第 1 版第 2 次印刷
开　　本：147mm × 210mm　1/32
印　　张：5.625
书　　号：ISBN 978-7-111-50207-4
定　　价：39.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066
购书热线：（010）68326294　88379649　68995259
投稿热线：（010）88379604
读者信箱：hzit@hzbook.com

The Translator's Words 译者序

今日之时代是大数据时代，大数据在各行各业中都发挥着巨大的作用。这对于广大开发者们也提出了新的要求，如何追赶技术的脚步、如何在大数据时代走在技术的前列成为摆在每一个技术人员面前的问题。提到大数据就不得不提到 Hadoop 以及围绕 Hadoop 形成的大数据生态系统：HBase、Hive、Pig、ZooKeeper、Flume、Storm 等。这其中更是以 Hadoop 为整个生态系统的核心和重中之重。虽然面临着后来者 Spark 的挑战，但 Hadoop 凭借成熟的工具库、完善的生态系统、业界的广泛应用依旧深受各大互联网公司的青睐。目前市场上关于 Hadoop 的技术图书已经汗牛充栋，那么，本书有哪些特点值得各位读者阅读呢？本书共包含两篇内容：上篇是关于 Apache Flume 的介绍；下篇是关于 MapReduce 模式的介绍。上篇针对 Apache Flume 进行了深入讲解。值得一提的是，目前市场上关于 Flume 的技术图书非常少，本书的出现很好地填补了这一空白。该篇首先从 Flume 的宏观架构谈起，接下来介绍如何安装与使用 Flume，然后对 Flume 的核心组件通道、接收器、源、拦截器等进行深入讲解，最后介绍 Flume 监控方面的知识并总结了实时分布式数据收集

的现状。相信各位读者可以从中学到有关 Apache Flume 方方面面的知识并能灵活地运用到项目中。对于 MapReduce 模式，下篇独辟蹊径，针对一个个问题分别进行介绍，摒弃了传统的流水账讲解方式，这样读者可以带着问题持续阅读，每一个案例最后都会给出相应的解决方案。这些案例都是与 Hadoop 相关的非常经典的案例，从最简单的使用 Java 编写一个单词统计应用到复杂的使用 MapReduce 实现 Kmeans，相信各位读者从中可以学到许多关于 Hadoop 的知识。

总的来说，这本涵盖 Flume 与 Hadoop 两个重要主题的图书会给各位读者带来耳目一新之感，同时这两个主题之间也存在着必然的联系，对有志于在大数据领域深耕的广大技术从业人员来说是一本上佳的领域专著。

翻译技术图书是一项艰苦的工作，首先非常感谢机械工业出版社华章公司的编辑们，感谢你们在图书翻译过程中给予我的支持与鼓励；此外，我还要对妻子张明辉说声感谢，你无微不至的关怀让我能够专心于翻译工作，同时也将本书献给我一岁多的孩子张梓轩，希望你在未来能够茁壮成长；最后，我要感谢我的父母，没有你们的养育就不会有今天的我。

尽管在翻译过程中本人已经付出了巨大的努力，但错误与遗漏之处在所难免，恳请广大读者将发现的错误及时告知于我以便在后续版本中能够及时更正。我的邮箱是 zhanglong217@163.com，新浪微博是 @ 风中叶的思考，欢迎关注。

张龙

2015 年 4 月

Preface 前　言

Hadoop是个非常优秀的开源工具，可以将海量的非结构化数据转换为易于管理的内容，从而更好地洞察客户需求。它很便宜（几乎是免费的），只要数据中心有空间和电源，它就能够水平扩展，并且可以处理传统数据仓库难以解决的问题。需要注意的是，你得将数据填入Hadoop集群中，否则你所得到的只不过是昂贵的热量产生器而已。你很快就会发现，一旦对Hadoop的使用经过“试验性”阶段后，你就需要工具来自动化地将数据填充到集群中。过去，你需要自己找到该问题的解决方案，但现在则不必如此！Flume一开始是Cloudera的项目，当时它们的集成工程师需要一次又一次地为客户编写工具来实现数据的自动化导入。时至今日，Flume已经成为Apache软件基金会的项目，并且处于活跃开发状态中，很多用户已经将其用于产品环境多年。

本书将会通过Flume的架构概览与快速起步指南帮助你迅速掌握Flume。接下来将会深入介绍Flume众多更加有用的组件的细节信息，包括用于即时数据记录持久化的重要的文件通道、用

于缓存并将数据写到 HDFS 中的 HDFS 接收器，以及 Hadoop 分布式文件系统。由于 Flume 内置很多模块，因此上手 Flume 时你所需要的唯一工具就是一个用于编写配置文件的文本编辑器而已。

通过阅读上篇，你将掌握构建高可用、容错、流式数据管道（用于填充 Hadoop 集群）所需的一切知识。

虽然网上关于 Hadoop 的资料已经汗牛充栋，但大多数都止步于表面或是仅针对某个具体问题给出解决方案。下篇则对 Hadoop 以及 MapReduce 编程进行了简明介绍，旨在让你快速起步并对 Hadoop 编程有个总体印象，打好基础才能深入探索每一类 MapReduce 问题。

本书主要内容

第 1 章介绍了 Flume 及其将会解决的问题空间（特别是与 Hadoop 相关的部分），架构概览介绍了将会在后续章节中讨论的各个组件。

第 2 章帮助你尽快上手 Flume，包括下载 Flume、创建“Hello World”配置并运行。

第 3 章介绍了大多数人都会用到的两个主要通道以及每个通道可用的配置选项。

第 4 章详细介绍了如何使用 HDFS Flume 输出，包括压缩选项以及数据格式化选项。此外还介绍了故障恢复选项以创建更为健壮

的数据管道。

第 5 章介绍了几种 Flume 输入机制及其配置选项。此外，还介绍了如何根据数据内容在不同的通道间切换，这样就可以创建复杂的数据流了。

第 6 章介绍了如何即时转换数据以及如何从负载中抽取信息来与通道选择器搭配以进行路由判定。还介绍了如何通过 Avro 序列化对 Flume 代理进行分层，如何将 Flume 命令行作为独立的 Avro 客户端进行测试以及手工导入数据。

第 7 章介绍了内外部用于监控 Flume 的各种方式，包括 Monit、Nagios、Ganglia 以及自定义钩子。

第 8 章超越了 Flume 配置与使用本身，对实时分布式数据收集的各个方面进行了讨论。

第 9 章介绍了如何使用 Java（不使用 MapReduce）编写一个单词统计应用。我们会将其与 MapReduce 模型进行比对。

第 10 章介绍了如何使用 MapReduce 编写单词统计应用以及如何使用 Hadoop 本地模式运行。

第 11 章介绍了如何在分布式环境中安装 Hadoop 并运行之前的 Wordcount job。

第 12 章介绍了如何编写一个 Hadoop 数据格式化器来读取 Amazon 数据格式以作为记录而非逐行读取数据。

第 13 章介绍了如何通过 MapReduce 处理 Amazon 数据，生成

直方图数据以及使用 gnuplot 来绘制结果。

第 14 章介绍了如何通过 MapReduce 连接两个数据集。

第 15 章介绍了如何处理 Amazon 数据以及通过 MapReduce 实现差集。此外还介绍了如何通过类似的方法实现其他的集合操作。

第 16 章介绍了如何通过 MapReduce 统计两个条目同时出现的次数（交叉相关）。

第 17 章介绍了如何处理 Amazon 数据以及通过反向索引实现简单的搜索。

第 18 章介绍了如何通过 MapReduce 实现图的遍历。

第 19 章介绍了如何通过 Kmeans 算法建立数据集的集群。集群会将数据划分为各个小组，这样每个分组中的条目都是类似的，根据不同的距离度量方法，不同分组中的条目是不同的。

学习本书的前提

你需要一台安装有 Java 虚拟机的 Linux 或 Mac 电脑，并安装有 JDK 1.6，因为 Flume 是用 Java 编写的。如果电脑上没有安装 Java，那么可以从 http://java.com/ 下载。

还需要网络连接，这样才能下载 Flume 并运行快速入门示例。

上篇主要讲解了 Apache Flume 1.3.0，此外还对 Cloudera 的 Flume CDH4 进行了介绍。

本书面向的读者

本书面向需要将来自各种系统的数据自动化地移动到 Hadoop 集群中的人。如果需要定期将数据加载到 Hadoop 中，那么本书就是为你准备的，它将使你从手工工作中解脱出来，也无需再继续维护一些自己编写的工具。

阅读本书只需要对基本的 Hadoop HDFS 知识有一定了解。我们提供了一些自定义的实现，如果需要可以使用。对于这种级别的实现，你需要了解如何使用 Java 进行编程。

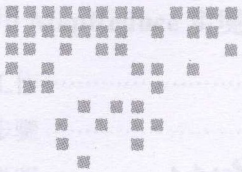
最后，你需要使用一款自己喜欢的文本编辑器，因为本书的大部分内容都涉及如何通过代理的文本配置文件来配置各种 Flume 组件。

目　录 *Contents*

下篇　MapReduce 模式

上篇 Part 1

Flume 日志收集



Chapter 1 第 1 章

概览与架构

如果在阅读本书，那就说明你正在数据的海洋中遨游。创建大量的数据是非常简单的事情，这要归功于 Facebook、Twitter、Amazon、数码相机与相机照片、YouTube、Google，以及你能想得到的能够连接到互联网上的任何东西。作为网站的提供者，10 年前的应用日志只是用来帮助你解决网站的问题。时至今日，如果你知道如何从大量的数据中浪里淘金，那么相同的数据就会提供关于业务与客户的有价值的信息。

此外，既然在阅读本书，那么你肯定知道创建 Hadoop 的目的在一定程度上就是为了解决大量数据的筛选问题。当然了，只有可靠地加载 Hadoop 集群数据并供数据科学家从中选择，这一切才能达成所愿。

将数据存储到 Hadoop 以及从 Hadoop 中获取数据（即 Hadoop

文件系统，HDFS）并不是什么难事——只需要如下一条命令即可：

```
% hadoop fs --put data.csv .
```

将数据打包好并准备上传时，使用上面这条命令就可以轻松将数据存储到 Hadoop 文件系统中。

不过，网站一直在创建着数据，批量将数据加载到 HDFS 中的频率是多少呢？每天？每小时？无论选择何种处理周期，最终还是会有人问“能否尽快给我数据呢”？你真正需要的是能够处理流式日志 / 数据的解决方案。

并不是只有你才有这种需求。Cloudera（专业的 Hadoop 服务提供商，拥有自己的 Hadoop 分发版本）在与客户的协作过程中不断发现了这种需求。创建 Flume 的目的就在于满足这种需求，它创建了一个标准、简单、健壮、灵活且可扩展的工具，用于将数据存储到 Hadoop 中。

1.1　Flume 0.9

Flume 是在 2011 年被首次引入到 Cloudera 的 CDH3 分发中的。它由一套工作守护进程（代理）构成，这些守护进程是通过 Zookeeper（一个配置与协调系统）根据一个或多个集中的 Master 配置而成的。在 Master 上，你可以在 Web UI 中查看代理状态，也可以以集中的方式在 UI 或是通过命令行 Shell 的方式取出配置（这两种方式都是通过 Zookeeper 与工作代理进行通信的）。

可以通过 3 种模式发送数据，分别叫作 Best Effort（BE）、Disk Failover（DFO）以及 End-to-End（E2E）。Masters 用于 E2E 模式，而多个 Master 配置尚不成熟，因此通常情况下只会使用一个 Master，

这使得其成为了E2E数据流失败的主要原因。Best Effort见名知意，代理会尝试并发送数据，如果无法发送，那么数据就会被丢弃。这种模式非常适合于度量等场景，一些差异是可以被接受的，因为新数据很快就会到来。DiskFailover模式会将无法发送的数据存储到本地磁盘上（有时也存储到本地数据库中），并且会不断重试，直到可以将数据发送到数据流中的下一个接受者为止。这对于计划好（或计划外）的断电场景很方便，只要有足够的本地磁盘能够缓存负载即可。

2011年6月，Cloudera将Flume项目的控制权交给了Apache基金会。2012年，Flume项目就从孵化状态变成了顶级项目。在孵化的这一年中，开发人员就已经开始基于Star Trek Themed标签对Flume进行重构，并创建了Flume-NG（Flume the Next Generation）。

1.2 Flume 1.X（Flume-NG）

Flume之所以会重构有很多原因，如果对细节感兴趣可以参考https://issues.apache.org/jira/browse/FLUME-728。一开始的重构分支最后变成了Flume 1.X的开发主线。

Flume 1.X最为明显的变化是不再使用中心化的配置Master/Masters与Zookeeper。Flume 0.9的配置有些过度烦琐，并且极易出错。此外，中心化的配置已经超出了Flume的目标范围。取代中心化配置的是一个简单的磁盘上的配置文件（不过配置文件是可插拔的，因此可以替换）。这些配置文件很容易通过诸如cf-engine、chef及puppet等工具分发。如果使用的是Cloudera分发包，那么可以通过Cloudera管理器来管理配置——最近其许可发生了变化，增加了节点限制，因此增加了吸引力。请确保不要手工管理这些配置，否

则就要一直采用手工方式编辑这些文件了。

Flume 1.X 的另一个主要差别是输入数据的读取与输出数据的写入现在由不同的工作线程（称为运行器）来处理了。在 Flume 0.9 中，输入线程也执行对输出的写入（故障恢复重试除外）。如果输出写入器很慢（而不仅仅是完全失败），那么它会阻塞 Flume 接收数据的能力。这种新的异步设计使得输入线程完全意识不到任何下游的问题。

本书介绍的 Flume 版本是 1.3.1（也是本书撰写之际 Flume 的当前版本）。

1.3　HDFS 与流式数据 / 日志的问题

HDFS 并不是真正的文件系统，至少从传统的认识来说不是这样，对于通常的文件系统来说，很多我们认为理所当然的东西并不适合于 HDFS，比如挂载。这使得将流式数据装载进 Hadoop 中变得有些复杂。

在通常的 Portable Operating System Interface（POSIX）风格的文件系统中，如果打开文件并写入数据，那么在文件关闭前它会一直存在于磁盘上。也就是说，如果另一个程序打开了相同的文件并开始读取，那么它会读取到写入器写到磁盘上的数据。此外，如果该写入进程中断，那么写到磁盘上的任何部分都是可用的（有可能不完整，但确实是存在的）。

在 HDFS 中，文件只作为目录项存在，在文件关闭前，其长度一直显示为 0。这意味着如果在一段时间内将数据写到文件中但却没有将其关闭，那么一旦客户端出现网络中断，你就什么都得不到了，只有一个空白文件而已。你会得出这样一个结论，即最好编写

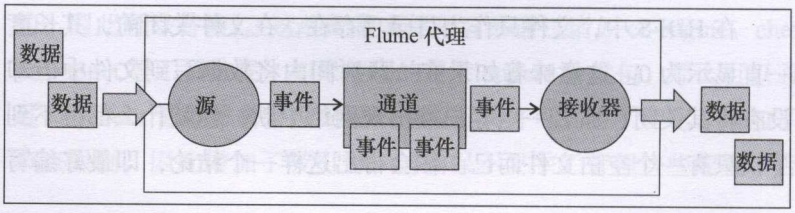
小文件，这样就能尽快将其关闭了。

问题在于 Hadoop 并不喜欢过多的小文件。由于 HDFS 元数据保存在 NameNode 的内存中，因此创建的文件越多，所需的 RAM 就越多。从 MapReduce 的角度来看，小文件会导致效率低下。通常情况下，每个 Mapper 都会被分配单个文件块作为输入（除非使用了某些压缩编码）。如果有过多的小文件，那么与待处理的数据相比，启动工作进程的代价就过高了。这种碎片还会导致更多的 Mapper 任务，使得总的 Job 运行时间增加。

在决定写入到 HDFS 的周期时需要考虑这些因素。如果计划保留数据较短的时间，那么可以使用较小的文件。然而，如果计划保留数据较长的时间，那么可以使用较大的文件或是做一些周期性的清理工作，将小文件压缩为少量的大文件，使得它们更加适合于 MapReduce。毕竟，你只是写入一次数据，但却要在这些数据上运行 MapReduce 任务成百上千次。

1.4 源、通道与接收器

Flume 代理的架构可以通过下面这张简单的图呈现出来。输入叫作源，输出叫作接收器。通道提供了源与接收器之间的胶水。它们都运行在叫作代理的守护进程中。



请记住如下概念：

源将事件写到一个或多个通道中。

通道作为事件从源到接收器传递的保留区。

接收器只从一个通道接收事件。

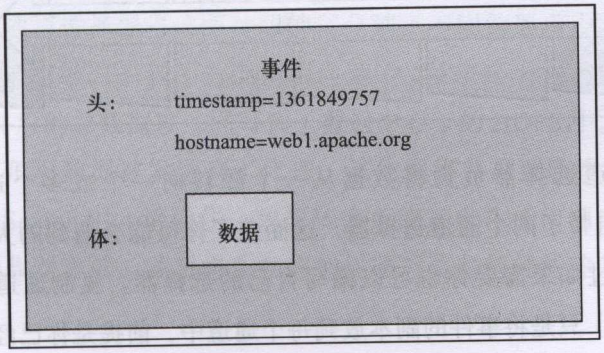
代理可能会有多个源、通道与接收器。

1.5　Flume 事件

Flume 传输的基本的数据负载叫作事件。事件由 0 个或多个头与体组成。

头是一些键值对，可用于路由判定或是承载其他的结构化信息（比如说事件的时间戳或是发出事件的服务器主机名）。你可以将其看作是与 HTTP 头完成相同的功能——传递与体不同的额外信息的方式。

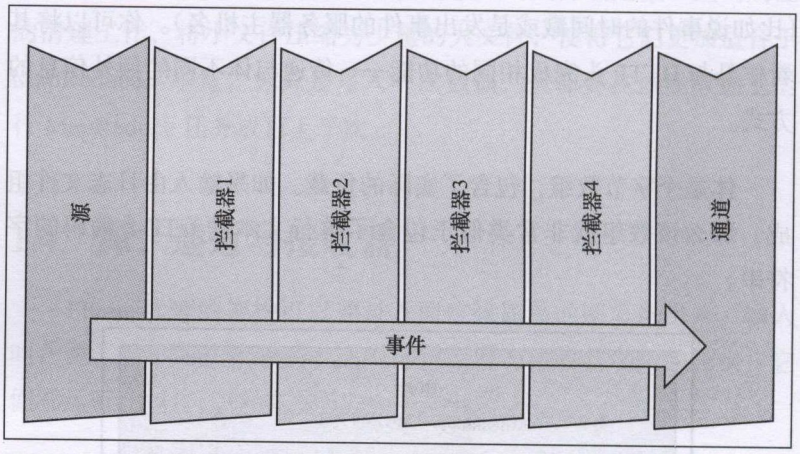
体是个字节数组，包含了实际的负载。如果输入由日志文件组成，那么该数组就非常类似于包含了单行文本的 UTF-8 编码的字符串。



Flume可能会自动添加头（比如，源添加了数据来自的主机名或是创建了事件时间戳），不过体基本上是不受影响的，除非你在中途使用拦截器对其进行了编辑。

1.5.1 拦截器、通道选择器与选择处理器

拦截器指的是数据流中的一个点，你可以在这里检查和修改Flume事件。你可以在源创建事件后或是接收器发送事件前链接0个或多个拦截器。如果熟悉AOP Spring框架，那么它非常类似于MethodInterceptor。在Java Servlets中，它类似于ServletFilter。在一个源上链接了4个拦截器，如下图所示。



通道选择器负责将数据从一个源转向一个或多个通道上。Flume自带了两个通道选择器，这涵盖了你可能会遇到的大多数场景。不过如果需要你也可以编写自己的选择器。复制通道选择器（默认的）只是将事件的副本放到每个通道中，前提是你已经配置好

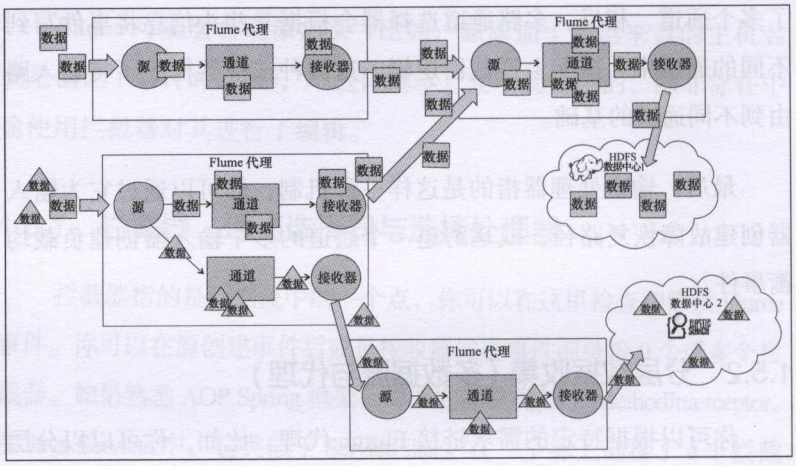
了多个通道。相反，多路通道选择器会根据某些头信息将事件写到不同的通道中。搭配上拦截器逻辑，这两种选择器构成了将输入路由到不同通道的基础。

最后，输入处理器指的是这样一种机制，你可以通过它为输入器创建故障恢复路径，或是跨越一个通道的多个输入器创建负载均衡事件。

1.5.2　分层数据收集（多数据流与代理）

你可以根据特定的需求链接 Flume 代理。比如，你可以以分层的方式插入代理来限制想要直接连接到 Hadoop 的客户端数量。很多时候，源机器没有足够的磁盘空间来处理长期停机或是维护窗口，这样就可以在源与 Hadoop 集群之间创建一个拥有大量磁盘空间的层次。

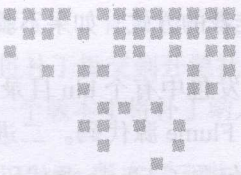
从下图中可以看到，数据在两个地方被创建（位于左侧），并且有两个最终目的地（位于右侧的 HDFS 与 ElasticSearch 云气泡）。下面增加点儿趣味性，假设有一台机器生成了两种数据（我们将其称作正方形数据与三角形数据）。我们在左下角的代理中使用了多路通道选择器将这两种数据划分到了不同的通道中。接下来，矩形通道被路由到了右上角的代理（以及来自于左上角的数据）。合并后的总数据被一同写到了数据中心 1 的 HDFS 中。与此同时，三角形数据被发送到了代理，该代理将其写到数据中心 2 的 ElasticSearch 中。请记住，数据转换可能发生在任何源之后以及任何接收器之前。随后将会介绍如何通过这些组件构建复杂的数据流。



1.6 小结

本章介绍了 Flume 尝试解决的问题，如何以一种易于配置且可靠的方式将数据加载进 Hadoop 集群中以进行数据处理。本章还介绍了 Flume 代理及其逻辑组件，包括事件、源、通道选择器、通道、接收处理器和接收器。

下一章将会更加详细地介绍这些主题，特别是每个组件最为常用的实现。就像其他优秀的开源项目一样，几乎所有这些组件都是可扩展的，如果它们无法满足你的需求，那么你可以自行对其进行扩展。



第 2 章 Chapter 2

Flume 快速起步

上一章已经介绍了关于 Flume 的一些基础知识，本章将会帮助你上手 Flume。首先从第一步开始，即下载和配置 Flume。

2.1 下载 Flume

请从 http://flume.apache.org/ 下载 Flume。在侧边导航栏找到下载链接，你会看到有两个压缩的 tar 归档，此外还有校验和与 gpg 签名文件，它们用于验证归档文件。网站上已经提供了验证下载文件的说明，因此这里就不再赘述了。针对实际的校验和检查校验和文件，以此验证下载的文件没有损坏。检查签名文件来验证下载的所有文件（包括校验和与签名）都来自于 Apache 而不是其他地方。你真的需要验证下载的文件么？一般来说，这是个好做法，

也是Apache推荐的方式。如果不验证，那我也没什么好说的。

二进制分发包中有个bin目录，源文件则位于src目录中。源文件只包含了Flume源代码。二进制分发包体积要更大一些，这是因为它不仅包含了Flume源代码与编译后的Flume组件（JARs、javadocs等），还包含了所有Flume依赖的Java库。二进制包包含了与源文件相同的Maven POM文件，因此如果从二进制分发包开始，那么你可以重新编译源代码。

继续，下载（并验证）二进制分发包，这样可以在起步时节省一些时间。

Hadoop分发包中的Flume

某些Hadoop分发包中自带了Flume。这些分发包可能提供了Hadoop核心组件包与相关项目（比如Flume），他们会考虑诸如版本兼容性与额外的Bug修复等问题。这些分发包没什么好坏之分，只是有些不同而已。

使用分发包有一些好处。因为之前已经有人将所有版本兼容的组件放到了一起。现在，这已经不是什么问题了，因为Apache Bigtop项目（http://bigtop.apache.org/）已经开始了。不过，预先构建好的标准OS包（如RPMs和DEBs等）省却了安装以及提供启动/关闭脚本的麻烦。每个分发包都提供了不同程度的付费选项，如果遇到了无法解决的问题，那么你可以购买付费的专业服务。

当然了，缺点还是有的。分发包自带的Flume版本常常落后于Apache发布的版本。如果有你想要使用的新特性，那要么等待分发包提供者进行移植，要么自己打补丁。此外，虽然分发包提供者做了大量的测试（就像任何通用平台一样），但你还是有可能会遇

到测试没有覆盖的问题。在这种情况下，你就得采取一些变通方式或是深入研究源代码了，还可以将补丁提交到开源社区（这样分发包就会在未来进行更新或是在下一个版本中将补丁纳入进来）。

因此，在 Hadoop 分发包中，Flume 版本会有些老。这种事情的好坏就见仁见智了。通常情况下，大公司不喜欢不稳定的新技术，也不喜欢频繁变化，因为变化可能是导致意外的最常见的原因。你很难找到使用最新 Linux 内核的公司，大部分使用的还是 Red Hat Enterprise Linux（RHEL）、CentOS、Ubuntu LTS 等旨在提供稳定性与兼容性的分发包。如果你是一家初创公司，构建下一代的互联网时尚圈，那么你可能需要这些新技术以在竞争中处于上风。

如果考虑使用分发包，那么请搜索一下，看看每一种分发包都提供了什么。记住，每一个分发包其实都希望你最后需要它们的企业服务，而这种服务并不便宜。自己好好想想吧。

下面是一个简短且不权威的列表，列出了现有的一些厂商，你可以了解一下：

- Cloudera:http://cloudera.com/
- Hortonworks:http://hortonworks.com/
- MapR:http://mapr.com/

2.2　Flume 配置文件概览

既然已经下载好了 Flume，下面来花点时间看看如何配置代理。

Flume 代理的默认配置提供者使用了一个简单的键值对的 Java 属性文件，你需要在启动时向代理传递一个参数。由于可以在单个文件

中配置多个代理，因此还需要额外传递一个代理标识符（叫作名字），这样它就知道该使用哪个代理了。在给出的示例中，我只指定了一个代理，使用agent这个名字。

每个代理的配置都以下面这3个参数开始：

```
agent.sources=<list of sources>
agent.channels=<list of channels>
agent.sinks=<list of sinks>
```

每个源、通道与接收器在该代理的上下文中也有一个唯一的名字。比如，如果不打算传递Apache访问日志，我就可以定义一个名为access的通道。该通道的配置都以前缀agent.channels.access开头。每个配置项都有一个type属性，告诉Flume是哪种源、通道还是接收器。在该示例中，我们使用一个内存通道，其类型为memory。名为agent的代理中的名为access的通道的完整配置如下所示：

```
agent.channels.access.type=memory
```

为源、通道与接收器指定的任何参数都会使用相同的前缀以额外属性的方式添加进来。memory通道有一个capacity参数，标识它能持有的最大的Flume事件数量。假如我们不想使用100这个默认值，那么配置文件将会如下所示：

```
agent.channels.access.type=memory
agent.channels.access.capacity=200
```

最后，我们需要将access通道名添加到agent.channels属性中，这样代理就会加载它了：

```
agent.channels=access
```

下面来看看使用标准“Hello World”的完整示例。

2.3　从“Hello World”开始

每一本技术图书都会有一个“Hello World”示例。下面是我们将会使用的配置文件：

```
agent.sources=s1
agent.channels=c1
agent.sinks=k1
agent.sources.s1.type=netcat
agent.sources.s1.channels=c1
agent.sources.s1.bind=0.0.0.0
agent.sources.s1.port=12345
agent.channels.c1.type=memory
agent.sinks.k1.type=logger
agent.sinks.k1.channel=c1
```

这里定义了一个名为 agent 的代理，它有一个名为 s1 的源、一个名为 c1 的通道，以及一个名为 k1 的接收器。

源 s1 的类型为 netcat，它只是打开一个 Socket 监听事件（每个事件一行文本）。它需要两个参数，分别是一个绑定 IP 与一个端口号。该示例使用 0.0.0.0 作为绑定地址（表示监听任何地址的 Java 约定）以及端口号 12345。源配置还有一个名为 channels 的参数，它是源将事件附加到的通道名。在该示例中使用的是 c1。这里使用了复数，因为你可以配置将一个源写到多个通道中，不过在这个简单的示例中我们并没有这么做。

名为 c1 的通道是个内存通道，使用了默认配置。

名为 k1 的接收器的类型是 logger。该接收器主要用于调试与测试。它会使用 log4j 将所有 INFO 级别的日志记录下来，这些日志来自于配置好的通道，在该示例中就是 c1。这里所用的通道是单数

的，因为一个接收器只能从一个通道接收数据。

使用该配置，我们运行代理并使用Linux netcat工具连接到代理来发送事件。

首先，解压缩之前下载的二进制分发包：

```
$ tar -zxf apache-flume-1.3.1.tar.gz
$ cd apache-flume-1.3.1
```

接下来，简单看看help命令。使用help命令运行flume-ng命令：

```
$ ./bin/flume-ng help
Usage: ./bin/flume-ng<command> [options]...

commands:
help                    display this help text
agent                   run a Flume agent
avro-client             run an avro Flume client
version                 show Flume version info

global options:
--conf,-c <conf>        use configs in <conf> directory
--classpath,-C <cp>     append to the classpath
--dryrun,-d             do not actually start Flume, just print the command
-Dproperty=value        sets a JDK system property value

agent options:
--conf-file,-f <file> specify a config file (required)
--name,-n <name>        the name of this agent (required)
--help,-h               display help text


avro-client options:
--dirname<dir>        directory to stream to avro source
```

```
--host,-H <host>          hostname to which events will be sent (required)
--port,-p <port>          port of the avro source (required)
--filename,-F <file>  text file to stream to avro source [default: std
input]
--headerFile,-R <file>headerFile containing headers as key/value pairs on
each

new line
--help,-h                 display help text
```

注意，如果指定了<conf>目录，那么就需要先将其加入到classpath中。

如你所见，可以通过两种方式调用命令（除了烦琐的help与version命令外）。我们将使用agent命令，avro-client的使用将在后面进行介绍。

agent命令有两个必填参数，分别是使用的配置文件与代理名（如果配置包含了多个代理）。下面来看看我们的示例配置，打开一个编辑器（我使用的是vi，你可以使用自己喜欢的任何编辑器）：

```
$ vi conf/hw.conf
```

接下来，将vi配置的内容放到编辑器中，保存并退回到shell。

现在可以启动代理了：

```
$ ./bin/flume-ng agent -n agent -c conf -f conf/hw.conf
-Dflume.root.logger=INFO,console
```

-Dflume.root.logger属性覆盖了conf/log4j.properties中的root

logger，使用console追加器。如果没有覆盖root logger，那么一切也都正常，只不过输出将会被写到log/flume.log文件中。当然了，你还可以编辑conf/log4j.properties文件，修改flume.root.logger属性（或是其他想要修改的任何属性）。

你可能会问，既然-f参数包含了配置的完整相对路径，那为何还要指定-c参数呢。原因在于log4j配置文件需要放到classpath中。如果命令没有指定-c参数，那就会报如下错误：

```
log4j:WARN No appenders could be found for logger
  (org.apache.flume.lifecycle.LifecycleSupervisor).
  log4j:WARN Please initialize the log4j system properly.
  log4j:WARN See
  http://logging.apache.org/log4j/1.2/faq.html#noconfig for more
  info.
```

不过你并没有这么做，因此会看到如下关键的日志输出：

```
2013-03-03 12:26:47,437 (main) [INFO -
  org.apache.flume.node.FlumeNode.start(FlumeNode.java:54)] Flume
  node starting - agent
```

这行日志告诉你名为agent的代理已经启动了。通常情况下，如果在配置文件中有多个配置，你只需要这一行日志就可以断定启动了正确的配置。

```
2013-03-03 12:26:47,448 (conf-file-poller-0) [INFO -
  org.apache.flume.conf.file.
  AbstractFileConfigurationProvider$FileWatcherRunnable.run
  (AbstractFileConfigurationProvider.java:195)] Reloading
  configuration file:conf/hw.conf
```

下面是另一个检查，确保加载了正确的文件，在该示例中就是hw.conf文件：

```
2013-03-03 12:26:47,516 (conf-file-poller-0) [INFO -
  org.apache.flume.node.nodemanager.DefaultLogicalNodeManager.
  startAllComponents(DefaultLogicalNodeManager.java:106)] Starting
  new configuration:{ sourceRunners:{s1=EventDrivenSourceRunner: {
  source:org.apache.flume.source.NetcatSource{name:s1,state:IDLE}
  }} sinkRunners:{k1=SinkRunner: {
  policy:org.apache.flume.sink
  .DefaultSinkProcessor@42552ccounterGroup:{ name:null counters:{}
  } }} channels:{c1=org.apache.flume.channel.MemoryChannel{name:
  c1}} }
```

当所有的配置都被解析完毕后，你会看到下面这条消息，它展示了配置好的一切内容。你会看到 s1、c1、k1，以及哪些 Java 类完成了实际的工作。你可能会猜想，netcat 其实是 org.apache.flume.source.NetcatSource 的便捷方式。如果需要，我们也可以使用类名。事实上，如果编写了自定义的源，那么我会使用其类名来作为源的 type 参数。如果不对 Flume 分发包打补丁，就无法定义自己的短名字：

```
2013-03-03 12:26:48,045 (lifecycleSupervisor-1-1) [INFO -
  org.apache.flume.source.NetcatSource.start
  (NetcatSource.java:164)] Created
  serverSocket:sun.nio.ch.ServerSocketChannelImpl[/0.0.0.0:12345]
```

我们看到源现在正在监听端口 12345 的输入，接下来向其发送一些数据。

最后，再打开一个终端。我们使用 nc 命令（也可以使用 telnet 或是其他类似的命令）发送字符串“Hello World”并敲入 <RETURN> 来标识事件的结束：

```
% nclocalhost 12345
  Hello World<RETURN>
  OK
```

“OK”来自于代理，在敲入回车后，它表示将文本行作为一个 Flume 事件来接受。如果查看代理日志，你会看到如下内容：

```
2013-03-03 12:39:58,582 (SinkRunner-PollingRunner-
  DefaultSinkProcessor) [INFO -
  org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)]
  Event: { headers:{} body: 48 65 6C6C6F 20 57 6F 72 6C 64
  Hello World }
```

上述日志消息表明该Flume事件不包含头（netcat源自己也没有添加任何头）。体以十六进制的形式呈现，并且还有一个字符串表示（便于阅读，该示例中就是Hello World消息）。

如果像下面这样再发送一行：

```
The quick brown fox jumped over the lazy dog.<RETURN>
  OK
```

你会在代理的日志中看到如下内容：

```
2013-03-03 12:45:08,466 (SinkRunner-PollingRunner-
  DefaultSinkProcessor) [INFO -
  org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)]
  Event: { headers:{} body: 54 68 65 20 71 75 69 63 6B 20 62 72 6F
  77 6E 20 The quick brown  }
```

事件似乎被截断了。根据设计，logger sink将体内容限制为16字节，从而避免屏幕充斥着过多的内容。如果想要查看调试的完整内容，那么你应该使用其他的sink，也许可以使用file_roll sink，它会将日志写到本地文件系统中。

2.4 小结

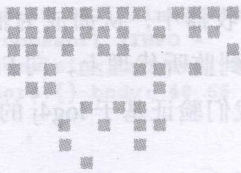
本章介绍了如何下载Flume二进制分发包。我们创建了一个简单的配置文件，里面包含了一个源，它会将内容写到一个通道中，后者又会将其写到一个接收器中。源监听着一个Socket，等待网络客户端的连接，并向其发送事件数据。这些事件被写到一个内存通

道中，然后被写到一个 log4j 接收器中，从而成为输出。接下来，我们使用 Linux netcat 工具连接到监听代理上，向 Flume 代理的源发送一些字符串事件。最后，我们验证基于 log4j 的接收器成功将事件写出。

下一章将会详细介绍在数据处理工作流中会用到的两个主要通道类型：

- 内存通道
- 文件通道

对于每一种类型的通道，我们都会介绍所有相关的配置，何时以及为何要覆盖默认值，更为重要的是，我们会介绍在何种场景下该使用哪一种通道。



Chapter 3 第 3 章

通　道

在 Flume 中，通道指的是位于源与接收器之间的构件。它为流动的事件提供了一个中间区域，从源中读取并且被写到数据处理管道中的接收器的事件处于这个区域中。

本章将要介绍的两类通道分别是内存 / 非持久化通道与本地文件系统 / 持久化通道。持久化文件通道会在发送者接收到事件前将所有变化写到磁盘上。它要比非持久化的内存通道慢一些，不过可以在出现系统事件或是 Flume 代理重启时进行恢复。与之相反，内存通道要更快一些，不过在出现失败时会导致数据丢失，并且与拥有大量磁盘空间的文件通道相比，它的存储能力要低很多。到底使用哪个通道取决于特定的用例、失败场景以及磁盘容忍度。

也就是说，无论选择哪一种通道，如果从源到通道的数据存储率

大于接收器所能写出的数据率，那就会超出通道的处理能力，并且会抛出 ChannelException 异常。源到底能对该 ChannelException 异常做什么以及不能做什么是特定于源本身的，不过在某些情况下，数据丢失是可能的，因此你需要恰当地规划好数据大小以避免填满通道。事实上，你总是希望接收器的写入速度要快于源的输入速度。否则就会出现这样一种情况，即一旦接收器失败了，那么你就再也追不上了。如果数据追踪的是站点使用情况，那么在白天时数据量就会很大，而在晚上时则会低一些，这样通道就有时间将数据传输出去了。事实上，你希望保持通道的深度（当前通道中的事件数量）尽可能低一些，这是因为在到达最终目的地之前，通道中所花费的时间会变成延迟时间。

3.1 内存通道

见名知意，内存通道指的是事件存储在内存中的通道。由于通常情况下，内存的速度要比磁盘快几个数量级，因此事件的接收速度也会更快，这降低了对硬件的需求量。使用这种通道的弊端在于代理失败（如硬件问题、断电、JVM 崩溃、Flume 重启等）会导致数据丢失。根据使用场景的不同，这可能是非常不错的解决方案。系统度量通常属于这一类，因为少量的数据丢失并不会造成什么影响。然而，如果事件表示的是网站的购买情况，那么内存通道就是一种非常差劲的选择了。

要想使用内存通道，请将通道的 type 参数设定为 memory。

```
agent.channels.c1.type=memory
```

上面为名为 agent 的代理定义了一个名为 c1 的内存通道。

下面是一个配置参数表格，你可以据此调整默认值：

键	是否必须	类型	默认值
type	是	String	memory
capacity	否	int	100
transactionCapacity	否	int	100
byteCapacityBufferPercentage	否	int（百分比）	20%
byteCapacity	否	long（字节）	JVM 堆大小的 80%
keep-alive	否	int	3 秒

该通道的默认容量是 100 个事件，这可以通过设置 capacity 属性来达成：

```
agent.channels.c1.capacity=200
```

记住一点，如果增加了这个值，那么你还需要增加 Java 堆空间大小，方式是使用 -Xmx 以及可选的 -Xms 参数实现。

你可以设定的另一个与容量相关的设置是 transactionCapacity。它指的是源的 ChannelProcessor（负责在单个事务中将数据从源移动到通道中的组件）可以写入的最大的事件数量。它也指的是 SinkProcessor（负责将数据从通道移动到接收器的组件）在单个事务中所能读取的最大的事件数量。你可以将这个值设得大一些，从而降低事务包装器的代价，这会提升速度。对于失败事件来说，增加这个值的弊端在于源需要回滚更多的数据。

Flume 只对每个独立代理中的每个通道提供了事务保证。在多代理中，多通道配置会出现重复，因此传输顺序可能会发生错乱，这并非正常情况。如果在非失败的情况下得到了重复，那就意味着你需要继续调节 Flume 的配置。

如果使用的接收器得益于更大规模的批量处理（比如 HDFS）将

内容写到某处，那么你应该将该值设得更大一些。就像其他东西一样，唯一能够确保的就是使用不同值来运行性能测试。Flume 提交者 Mike Percy 所写的博文 http://bit.ly/flumePerfPt1 值得你好好读读。

https://issues.apache.org/jira/browse/FLUME-1535 介绍了 byteCapacityBufferPercentage 与 byteCapacity 参数，它们使用字节而非事件数量来作为调整内存通道大小的方式，同时还避免了 OutOfMemoryErrors。如果事件大小的变化范围很大，那么你应该使用这些设置来调整容量，但需要注意的是，计算只是根据事件体来评估的。如果有任何头，那么实际的内存使用量就会比配置的值大一些。

最后，keep-alive 参数指的是通道已满并且在放弃前，线程将数据写到通道中的等待时间。由于数据同时也在从通道中被写出，因此如果在超时到期后又有了新的空间，那么数据还是会被写入通道中，而不会向源抛出异常。你可能想将该值设得非常大，不过请记住，等待写入到通道会阻塞数据进入到源中，这可能会导致数据堵在上游代理中。最后，这会造成事件被丢弃。你需要针对周期性的流量变化以及临时计划（或未计划）的维护情况进行调整。

3.2 文件通道

文件通道指的是将事件存储到代理本地文件系统中的通道。虽然要比内存通道慢一些，不过它却提供了持久化的存储路径，可以应对大多数情况，它应该用在数据流中不允许出现缺口的场合。

这种持久化能力是由 Write Ahead Log（WAL）以及一个或多个文件存储目录联合提供的。WAL 用于以一种原子且安全的方式追踪来自于通道的所有输入与输出。通过这种方式，如果代理重启，那么 WAL 可以重放，从而确保在清理本地文件系统的数据存储前进

入到通道中的所有事件都会被写出。

此外，如果数据处理策略要求磁盘上的所有数据（甚至是临时数据）都要加密，那么文件通道还支持将加密数据写到文件系统中。本书并不会对此进行介绍，不过如果需要，那么可以看看 Flume 用户指南（http://flume.apache.org/FlumeUserGuide.html）上的示例。请记住，使用加密会降低文件通道的吞吐量。

要想使用文件通道，请将通道的 type 参数设定为 file：

```
agent.channels.c1.type=file
```

上面为名为 agent 的代理定义了一个名为 c1 的文件通道。

下面是一个配置参数表格，你可以据此调整默认值：

键	是否必须	类型	默认值
type	是	String	file
checkpointDir	否	String	~/.flume/file-channel/checkpoint
dataDirs	否	String（逗号分隔的列表）	~/.flume/file-channel/data
capacity	否	int	1000000
keep-alive	否	int	3 秒
transactionCapacity	否	int	1000
checkpointInterval	否	long	300000 毫秒，即 5 分钟
write-timeout	否	int	10 秒
maxFileSize	否	long	2146435071 字节
minimumRequiredSpace	否	long	524288000 字节

要想指定 Flume 代理持有数据的位置，你需要设定 checkpointDir 与 dataDirs 属性：

```
agent.channels.c1.checkpointDir=/flume/c1/checkpoint
agent.channels.c1.dataDirs=/flume/c1/data
```

从技术上来说，这些属性并不是必需的，对于开发来说，其默认值已经很不错了。不过，如果代理中配置了多个文件通道，那么只有第一个通道会启动。对于使用了多个文件通道的产品部署与开发来说，你应该针对每个文件通道存储区域使用不同的目录路径，并考虑将不同的通道放在不同的磁盘上以避免 IO 争抢。此外，如果使用大容量的机器，那么请考虑使用某种形式的包含了条带化的 RAID（RAID 10、50、60）来获得更好的磁盘性能，而不是购买更昂贵的 10k 或 15k 驱动器或是 SSDs。如果没有 RAID 条带化但却有多个磁盘，那么请将 dataDirs 设为包含每个存储位置的逗号分隔的列表。使用多个磁盘会分散磁盘访问，类似于条带化的 RAID，不过却没有 RAID 50/60 的计算代价以及 RAID 10 的 50% 的空间浪费。你可以测试一下系统，看看 RAID 的代价是否能够弥补这种速度上的差异。既然硬盘驱动器的故障是存在的，那么你就应该首选某些 RAID 配置而不是单独的磁盘，从而防止出现与单个驱动故障相关的数据丢失情况的发生。

出于相同的原因，你不应该使用 NFS 存储。使用 JDBC 通道是个不好的做法，因为它会引入瓶颈和单点故障，这些都是高性能分布式系统在设计上应该极力避免的。

请确保在使用文件通道时设定好了 HADOOP_PREFIX 与 JAVA_HOME 环境变量。虽然我们似乎并没有使用过任何 Hadoop 的东西（比如写入 HDFS 中），不过文件通道使用 Hadoop Writeables 作为一种磁盘序列化格式。如果 Flume 找不到 Hadoop 库，那么你会在启动时看到下面的异常，因此请检查环境变量设置：

java.lang.NoClassDefFoundError: org/apache/hadoop/io/Writable

默认的文件通道容量是 100 万个事件，无论事件内容的大小是多少均如此。如果到达了通道容量，那么源就无法再接收数据了。这个默认值对于低容量的场景来说还不错。如果接收的数据量很大，导致无法忍受正常规划或未规划的中断，那么你可以调整该值。比如，Hadoop 中有很多配置改变需要重启集群。如果 Flume 向 Hadoop 中写入的是一些重要数据，那么文件通道就应该调整以接受 Hadoop 重启的时间（或许对于意外的情况还需要添加一个缓冲）。如果集群或是其他系统是不可靠的，那么你可以将这个值设得更高一些来处理更长的宕机时间。有时，你会遇到磁盘空间不够用的情况，因此请设定好一个上限值（或是购买更大的磁盘）。

keep-alive 参数类似于内存通道。它指的是满载的通道在放弃前，源尝试写入到通道中的最大等待时间。如果超时前又有可用空间了，那么写入就将成功；否则会向源抛出 ChannelException 异常。

属性 transactionCapacity 指的是在单个事务中所允许的最大事件数量。对于某些源来说，它们会将事件集中到一起，然后在单个调用中传递给通道，对于这种情况来说，该参数就很重要。很多时候，你无需改变其默认值。将该值设得比较高会在内部分配额外的资源，因此除非遇到了性能问题，否则不应该增加其值。

checkpointInterval 属性指的是两个检查点（还会将日志文件写到 logDirs 中）之间间隔的毫秒数，不能将该值设为低于 1000 毫秒。

检查点文件也会使用 maxFileSize 属性根据写入到其中的文件量进行卷动。如果想节省一些磁盘空间，那么你可以针对低负载的通道降低该值的大小。假如最大的文件大小是 50000 字节，但通道一天只会写入 500 字节，那么要想填充满单个日志需要 100 天的时间。假如在

第100天的时候写入了2000字节，那么有些数据就会被写到旧的日志文件中，新的文件则从剩余部分开始。在卷动后，Flume会尝试删除不再需要的日志文件。由于完整的日志有未被处理的记录，因此还不能将其删除。下一次清理旧日志文件可能还得100天之后。如果旧的50000字节文件留存的时间更长，那就没什么问题，不过由于默认大小是2GB左右，因此每个通道所用的磁盘空间是其两倍（4GB）。取决于可用的磁盘空间大小以及代理中配置的通道数量，这可能是个问题，也可能不是什么问题。如果机器有大量的存储空间，那么使用其默认值就可以了。

最后，minimumRequiredSpace属性指的是你不想用作日志的空间数量。如果尝试使用与dataDir路径相关联的最后500MB磁盘空间，那么默认配置就会抛出异常。这个限制适用于所有通道，因此如果配置了3个文件通道，那么其上限依然是500MB而不是1.5GB。你可以将其值设为1MB那么小，不过一般来说，如果磁盘使用率接近于100%，就会出现一些不好的结果。

3.3 小结

本章介绍了在数据处理管道中常用的两类通道。

内存通道提供了更快的速度，这是以故障事件出现时数据丢失为代价的。

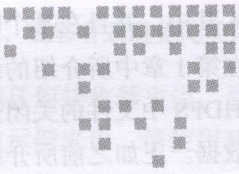
此外，文件通道提供了更可靠的传输，因为它能容忍代理故障与重启，这是以牺牲性能为代价的。

你需要确定哪种通道更适合于你的使用场景。在确定内存通道

是否适合时，请问问自己丢失一些数据的经济上的代价如何。在考虑是否使用持久化通道时请衡量它与添加更多的硬件以弥补性能上的差异时的代价相比如何。另一个考虑就是数据问题了。写入到Hadoop中的数据不一定都来自于流式应用日志。如果接收的是每天的数据下载，那么就可以使用内存通道了，因为一旦遇到了问题还可以重新导入。

接收流式数据时出现重复事件是有可能的。有些人会运行周期性的MapReduce job来清理数据（删除重复的事件）。其他人则会在运行自己的MapReduce job时才考虑重复，这样可以省去额外的后期处理时间。事实上，你可以同时采用这两种方式。

下一章将会介绍接收器。特别是将事件写到HDFS中的HDFS接收器；此外，还会介绍事件序列化器，它指定了如何将Flume事件转换为更加适合于接收器处理的输出。最后，下一章将会介绍接收处理器以及如何在分层配置中创建负载均衡与故障路径，从而实现更为健壮的数据传输。



第 4 章 Chapter 4

接收器与接收处理器

到目前为止，你应该已经非常清楚接收器在 Flume 架构中的位置了。本章将会介绍 Hadoop 最常使用的接收器——HDFS 接收器。Flume 的架构还支持很多其他类型的接收器，不过本书没有对其进行太多介绍。Flume 自带的一些接收器可以写到 HBase、IRC 及 ElasticSearch 中，本书第 2 章还介绍了 log4j 与文件接收器。网上还有其他一些接收器，可以将数据写到 MongoDB、Cassandra、RabbitMQ、Redis 以及其他数据存储中。如果找不到适合需求的接收器，你还可以通过继承 org.apache.flume.sink.Abstractsink 类轻松地写一个出来。

4.1 HDFS 接收器

HDFS 接收器的作用是持续打开 HDFS 中的文件，然后以流的

方式将数据写入其中，并且在某个时间点关闭该文件再打开新的文件。正如我们在第1章中所介绍的那样，文件之间的卷动持续多长时间一定要与HDFS中文件的关闭速度协调一致，这样才能以可视化的形式处理数据。正如之前所介绍的那样，输入中有太多的小文件会导致MapReduce job非常低效。

要想使用HDFS接收器，请将接收器的type参数设定为hdfs：

```
agent.sinks.k1.type=hdfs
```

上面一行代码为名为agent的代理定义了一个名为k1的HDFS接收器。你还需要指定其他一些必填参数，首先要指定的是HDFS中的path，它表示将数据写到的位置：

```
agent.sinks.k1.hdfs.path=/path/in/hdfs
```

类似于Hadoop中的大多数文件路径，该HDFS路径可以通过3种方式指定，分别是绝对路径、带有服务器名的绝对路径以及相对路径。它们是等价的（假定以Flume用户的身份运行Flume Agent）：

类型	路径
绝对路径	/Users/flume/mydata
带有服务器名的绝对路径	hdfs://namenode/Users/flume/mydata
相对路径	mydata

我喜欢通过hadoop命令行来配置安装Flume的机器，方式是在Hadoop的core-site.xml文件中设置fs.default.name属性。我不会在Hadoop用户目录中保存持久化数据，而是通过一些有意义的路径名来使用绝对路径（比如说/logs/apache/access）。只有在目标是完全不同的Hadoop集群时我才会特别指定一个name node。这样就可以将一个环境下测试过的配置迁移到另一个环境中而不会导致意外的后果，比如生产服务器将数据写入中间Hadoop集群中，只是因为有人忘记在配置中编辑目标。

具体化环境细节信息是个最佳实践，能够有效避免此类情况的发生。

对于 HDFS 接收器以及其他任何接收器来说，最后一个必填参数就是进行操作的通道。对于该示例来说，请使用要从中读取信息的通道名来设置 channel 参数：

```
agent.sinks.k1.channel=c1
```

上述代码告诉 k1 接收器从 c1 通道中读取事件。

下表列出了用于调整默认值的几乎所有的配置参数：

键	是否必须	类型	默认值
type	是	String	hdfs
channel	是	String	
hdfs.path	是	String	
hdfs.filePrefix	否	String	FlumeData
hdfs.fileSuffix	否	String	
hdfs.maxOpenFiles	否	long	5000
hdfs.round	否	Boolean	false
hdfs.roundValue	否	int	1
hdfs.roundUnit	否	String（秒、分钟或小时）	秒
hdfs.timeZone	否	String	本地时间
hdfs.inUsePrefix	否	String	（CDH4.2.0 或　是 Flume 1.4）
hdfs.inUseSuffix	否	String	.tmp（CDH4.2.0 或　是 Flume 1.4）
hdfs.rollInterval	否	long（秒）	30 秒（0 表示禁用）
hdfs.rollSize	否	long（字节）	1024 字节（0 表示禁用）
hdfs.rollCount	否	long	10（0 表示禁用）
hdfs.batchSize	否	long	100
hdfs.codeC	否	String	

请不要忘记在 http://flume.apache.org/ 上阅读 Flume 用户指南了

解所用的Flume版本，因为本书所介绍的版本与你实际使用的版本可能会有所差别。

4.1.1 路径与文件名

每次Flume在HDFS中的hdfs.path开启一个新文件写入数据时，该文件名都由hdfs.filePrefix、一个点符号、文件开启的时间戳以及由hdfs.fileSuffix属性指定的一个文件后缀构成（如果设置的话），请看下面这行代码：

```
agent.sinks.k1.hdfs.path=/logs/apache/access
```

上面这行代码会生成一个诸如/logs/apache/access/Flume Data.1362945258的文件。

不过，请看看下面的配置：

```
agent.sinks.k1.hdfs.path=/logs/apache/access
agent.sinks.k1.hdfs.filePrefix=access
agent.sinks.k1.hdfs.fileSuffix=.log
```

在该配置中，文件名更加类似于/logs/apache/access/access.1362945258.log。

随着时间的流逝，hdfs.path目录将会变满，因此你需要向路径中添加某种时间元素将文件划分为若干子目录。Flume支持各种基于时间的转义序列，如%Y表示由4个数字构成的年份。我喜欢年/月/天/小时形式的序列（根据最老到最新进行排序），因此我常常使用它来表示路径：

```
agent.sinks.k1.hdfs.path=/logs/apache/access/%Y/%m/%D/%H
```

上面的代码表示/logs/apache/access/2013/03/10/18/这样的路径。

要想了解完整的基于时间的转义序列列表，请参考 Flume 用户指南。

另一个非常便捷的转义序列机制是在路径中使用 Flume 头值的功能。比如，如果有个键为 logType 的头，那么我就可以将 Apache 访问日志与错误日志划分到不同目录中，同时还使用相同的通道，方式是像下面这样对头的键进行转义：

```
agent.sinks.k1.hdfs.path=/logs/apache/%{logType}/%Y/%m/%D/%H
```

这样，访问日志就会写到 /logs/apache/access/2013/03/10/18 中，而错误日志则会写到 /log/apache/error/2013/03/10/18/ 中。然而，如果想将这两种类型的日志写到相同的目录路径中，那么可以在 hdfs.filePrefix 中使用 logType，如下所示：

```
agent.sinks.k1.hdfs.path=/logs/apache/%Y/%m/%D/%H
agent.sinks.k1.hdfs.filePrefix=%{logType}
```

显然，Flume 可以一次写入多个文件。属性 hdfs.maxOpenFiles 设定了一次可以写多少的上限，默认值为 5000。如果超过了这个限制，那么处于打开状态的最老的文件将会被关闭。请记住，每个打开的文件在 OS 层次与 HDFS 层次（NameNode 与 DataNode 连接）上都会有一定的代价。

还有其他一些有用的属性，它们考虑到了事件时间在小时、分钟以及秒粒度上的舍入，同时又在文件路径上维护了这些元素信息。假如有下面的路径规范：

```
agent.sinks.k1.hdfs.path=/logs/apache/%Y/%m/%D/%H%M
```

这次你希望每天只有 4 个子目录（分别在每小时的 00、15、30 以及 45 分，每个都包含了 15 分钟的数据）。可以通过设置如下值来实现：

```
agent.sinks.k1.hdfs.round=true
agent.sinks.k1.hdfs.roundValue=15
agent.sinks.k1.hdfs.roundUnit=minute
```

这会将2013-03-10这一天01:15:00到01:29:59之间生成的日志写到/logs/apache/2013/03/10/0115/的文件中。01:30:00到01:44:59之间生成的日志将会写到/logs/apache/2013/03/10/0130/的文件中。

hdfs.timeZone属性用于指定转义序列中解释时间的时区，其默认值是计算机的本地时间。如果本地时间受夏时制影响，那么当%H==02时（秋天），数据将会变成两倍，当%H==02时（春天），将没有数据。我认为将时区引入到计算机所读取的数据上是非常不好的做法。我觉得时区只是人应该考虑的事情，计算机之间只需要基于国际标准时间进行通信。出于这个原因，我在自己的Flume代理上是像下面这样设置该属性的，从而消除了时区问题：

```
-Duser.timezone=UTC
```

如果不认同我的做法，那么你可以使用默认值（本地时间），或是将hdfs.timeZone设置为其他任何你想设置的值。你所传递的值会被用在对java.util.Timezone.getTimeZone(…)的调用上，因此请查看Javadocs了解这里可以使用的值。

最后，当文件被写到HDFS中时会添加一个.tmp扩展名。当文件关闭时，该扩展名会被删除。这样，当Flume频繁向目录中写入时，如果在该目录上运行MapReduce job，那么你就可以轻松地在输入阶段排除掉这些文件了。此外，还可以通过查看HDFS中的目录列表了解哪些文件被写入了。由于在MapReduce job中通常会指定输入目录（或是因为使用了Hive），因此你可能会错误地将这些临时文件当作空输入或是错误输入。基于此，FLUME-1702就是为了解决这个问题，并且会在Flume 1.4中发布，如果碰巧使用了Cloudera的CDH4.2.0版本，那

么这个补丁会被纳入到 Flume 1.3 中。这就引入了两个新属性来改变使用的前缀与后缀。为了避免在文件关闭前使用临时文件，请将后缀设为空（而不是默认的 .tmp），将前缀设为一个点或是一个下划线，如下所示：

```
agent.sinks.k1.hdfs.inUsePrefix=_
agent.sinks.k1.hdfs.inUseSuffix=
```

4.1.2　文件转储

默认情况下，Flume 会每隔 30 秒、10 个事件或是 1024 字节来转储写入的文件。这分别是通过设置 hdfs.rollInterval、hdfs.rollCount 与 hdfs.rollSize 属性实现的。你可以将这些属性值设为 0，从而禁用特定的转储机制。比如，如果只希望每分钟转储一次，那么应该像下面这样来设置这些参数：

```
agent.sinks.k1.hdfs.rollInterval=60
agent.sinks.k1.hdfs.rollCount=0
agent.sinks.k1.hdfs.rollSize=0
```

如果输出包含了头信息，那么每个文件的 HDFS 大小就要比期望的大一些，这是因为 hdfs.rollSize 转储模式只会考虑事件体的长度。显然，你不会同时禁用这三种转储机制，否则 HDFS 中就只有一个目录了，里面是变得越来越大的文件。

最后，一个相关参数是 hdfs.batchSize。它指的是在每个事务中接收器从通道所读取的事件数量。如果通道中有大量的数据，那么将该参数设为大于 100（默认值）会提升性能，这会降低每个事件的事务代价。

既然已经介绍了 HDFS 管理与归档文件的方式，下面就来看看事件内容是如何写入的。

4.2 压缩编解码器

编解码器用于通过各种压缩算法来压缩和解压缩数据。Flume支持gzip、bzip2、lzo和snappy，不过你需要自己安装lzo，特别是在使用了CDH分发版本的情况下，这是由于许可问题造成的。

如果希望HDFS接收器写入的是压缩文件，那么你需要指定数据的压缩方式，这是通过设置hdfs.codeC属性实现的。该属性还用作写入到HDFS中的文件的前缀。比如，如果像下面这样指定了编解码器，那么所有写入的文件都会带有一个.gzip扩展，因此在这种情况下就无需指定hdfs.fileSuffix属性了：

```
agent.sinks.k1.hdfs.codeC=gzip
```

选择使用哪个编解码器需要做些调研。对于使用gzip还是bzip2存在一些争论，因为它们拥有更高的压缩率，但压缩时间却更长，特别是数据写一次而要读成百上千次的情况。另一方面，使用snappy或lzo的压缩性能会更高，不过压缩率却降低了。请记住，文件的分割性（特别是使用纯文本文件）会极大地影响MapReduce job的性能。如果不了解这方面的情况，请看看《Hadoop Beginner's Guide》(http://amzn.to/14Dh6TA）或是《Hadoop: The Definitive Guide》(http://amzn.to/16OsfIf）吧。

4.3 事件序列化器

事件序列化器指的是Flume事件转换为另一种格式以进行输出的机制。它在功能上类似于log4j的Layout类。在默认情况下，text序列化器只会输出Flume事件体。还有一个header_and_text序列化器，它既会输出头信息也会输出体信息。最后，还有一个avro_event

序列化器，它可以创建事件的 Avro 表示。如果编写自己的序列化器，那么你需要使用实现的完整限定类名作为序列化器的属性值。

4.3.1　文本输出

如前所述，默认的序列化器是 text 序列化器，它只会输出 Flume 事件体而丢弃掉头信息。每个事件后面都有个换行符，除非将 serializer.appendNewLine 属性设为 false 来覆盖其默认行为。

键	是否必须	类型	默认值
serializer	否	String	text
serializer.appendNewLine	否	boolean	true

4.3.2　带有头信息的文本

你可以通过 text_with_headers 序列化器保存 Flume 事件头而不是将其丢弃。其输出格式包含了头信息，后跟一个空格，然后是体信息负载，最后以一个可选的禁用新行字符终止。下面是该序列化器所生成的输出示例：

```
{key1=value1, key2=value2} body text here
```

键	是否必须	类型	默认值
serializer	否	String	text_with_headers
serializer.appendNewLine	否	boolean	true

4.3.3　Apache Avro

Apache Avro 项目（http://avro.apache.org/）提供了一种类似于 Google protocol buffers 的序列化格式，不过对于 Hadoop 更加友好，因为其容器是基于 HadoopSequenceFiles 的，并且对 MapReduce 做了一些集成。其格式也是自我描述的，使用了 JSON，这有助于作为

一种长期的数据存储格式，因为数据格式可能会随着时间的推移而发生变化。如果数据存在大量的结构，而你又不想将其转换为String，只是希望在MapReduce job中才解析这些String，那么你应该更多地了解一下Avro，看看是否应该在HDFS中使用它来作为存储格式。

avro_event序列化器根据Flume事件模式来创建Avro数据。它没有格式化参数，因为Avro规定了数据格式，并且Flume事件结构规定了所用的模式：

键	是否必须	类型	默认值
serializer	否	String	avro_event
serializer.compressionCodec	否	String（gzip、bzip2、lzo或snappy）	
serializer.syncIntervalBytes	否	int（字节）	2048000（字节）

如果想要使用Avro，不过使用的模式与Flume事件模式不同，那就需要编写自己的事件序列化器了。

如果希望在将数据写到Avro容器前对其进行压缩，那么你应该将serializer.compressionCodec属性设为已安装的编解码器的文件扩展。serializer.syncIntervalBytes属性用于确定将数据推入到HDFS前所使用的数据缓冲大小，因此在使用编解码器时该设置会影响到压缩率。下面这个示例使用了4MB的缓冲对Avro数据进行snappy压缩：

```
agent.sinks.k1.serializer=avro_event
agent.sinks.k1.serializer.compressionCodec=snappy
agent.sinks.k1.serializer.syncIntervalBytes=4194304
agent.sinks.k1.hdfs.fileSuffix=.avro
```

要想让Avro文件能够用在Avro MapReduce job中，这些文件必须要以.avro结尾或是在输入时将其忽略掉。出于这个原因，你需要显式设置hdfs.fileSuffix属性。此外，你不能设置Avro文件的hdfs.codeC属性。

4.3.4　文件类型

默认情况下，HDFS 接收器会以 Hadoop SequenceFiles 的形式将数据写到 HDFS 中。这是个常见的 Hadoop 包装器，包含了一个键与一个值域，其中值域通过二进制字段与记录分隔符进行分隔。通常情况下，计算机上的文本文件会假定换行符确定了每一条记录。那么如果数据包含了换行符（如 XML），那该怎么办呢？使用序列化文件可以解决这个问题，因为它使用了不可打印的字符作为分隔符。SequenceFiles 也是可以分割的，这样在数据上（特别是大文件上）运行 MapReduce job 时就可以实现更好的定位与并行处理。

1. 序列文件

如果使用 SequenceFile 文件类型，那么你需要指定如何将键与值写到 SequenceFile 的记录中。每个记录的键总是一个 LongWritable，它包含了当前的时间戳，如果事件头设定了时间戳，那就会使用这个时间戳。默认情况下，值的格式是个与 byte[] Flume 体相对应的 org.apache.hadoop.io.BytesWritable：

键	是否必须	类型	默认值
hdfs.fileType	否	String	SequenceFile
hdfs.writeType	否	String	writable

不过，如果希望将负载解析为 String，那么你应该覆写 hdfs.writeType 属性，将 org.apache.hadoop.io.Text 作为值域：

键	是否必须	类型	默认值
hdfs.fileType	否	String	SequenceFile
hdfs.writeType	否	String	text

2. 数据流

如果由于数据没有自然的键而不想输出为SequenceFile，那么你可以使用DataStream只输出未压缩的值。这只要覆写hdfs.fileType属性即可：

```
agent.sinks.k1.hdfs.fileType=DataStream
```

该文件类型应该与Avro序列化搭配使用，因为任何的压缩操作都应该在事件序列化器中完成。要想序列化gzip压缩的Avro文件，你应该设置如下属性：

```
agent.sinks.k1.serializer=avro_event
agent.sinks.k1.serializer.compressionCodec=gzip
agent.sinks.k1.hdfs.fileType=DataStream
agent.sinks.k1.hdfs.fileSuffix=.avro
```

3. 压缩流

CompressedStream类似于DataStream，只不过其数据在写入时会被压缩。你可以将其看作是在一个未压缩的文件上运行gzip工具，不过所有操作都是一步完成的。它与压缩的Avro文件不同，后者的内容是先被压缩，然后写到未压缩的Avro包装器中。

```
agent.sinks.k1.hdfs.fileType=CompressedStream
```

如果决定使用CompressedStream，那么请记住，在MapReduce中只有某些压缩格式才能进行分割。压缩算法的选择并没有Flume配置，它是由核心Hadoop中的zlib.compress.strategy与zlib.compress.level属性确定的。

4.3.5　超时设置与线程池

最后，还有两个与超时相关的属性以及两个与线程池相关的属性：

键	是否必须	类型	默认值
hdfs.callTimeout	否	long（毫秒）	10000
hdfs.idleTimeout	否	int（秒）	0（0 表示禁用）
hdfs.threadsPoolSize	否	int	10
hdfs.rollTimerPoolSize	否	int	1

hdfs.callTimeout 属性指的是 HDFS 接收器在 HDFS 操作放弃前返回成功或是失败的时间。如果 Hadoop 集群速度很慢（比如开发或是虚拟化集群），那么你需要将这个值设得高一些以避免错误。请记住，如果写的吞吐比通道的输入速度低，那么通道就会溢出。

如果将 hdfs.idleTimeout 属性设为一个非 0 值，那么它指的就是 Flume 等待自动关闭空闲文件的时间。我从来没有使用过该属性，因为 hdfs.fileRollInterval 会处理每个周期中文件的关闭，如果通道空闲，它就不会再打开新的文件。该设置的目的似乎在于作为之前介绍过的大小、时间与事件数机制的一种替代方案。你希望尽可能多地向文件中写入数据，只在没有更多的数据可写时才关闭它。在这种情况下，如果将 hdfs.rollInterval、hdfs.rollSize 与 hdfs.rollCount 都设为 0，那么你可以通过 hdfs.idleTimeout 实现归档模式。

可用于调整线程数的第一个属性是 hdfs.threadsPoolSize，其默认值为 10。它指的是同一时刻可以写入的最大文件数。如果在确定文件路径与名字时使用了事件头，那么同一时刻打开的文件数就会超过 10 个，在调高该值时请务必小心行事，以免压垮 HDFS。

与线程池相关的最后一个属性是 hdfs.rollTimerPoolSize，它指的是处理由 hdfs.idleTimeout 属性所设定的超时的线程数。用于关闭文件的线程数非常少，因此你不太可能增加一个线程数这个默认

值。如果不使用基于hdfs.idleTimeout的归档，那就可以忽略掉hdfs.rollTimerPoolSize属性，因为用不上。

4.4 接收器组

为了在数据处理管道中消除单点失败，Flume提供了通过负载均衡或是故障恢复机制将事件发送到不同接收器的能力。为了做到这一点，我们需要引入一个叫作接收器组的新概念。接收器组用于创建逻辑接收器分组，该分组的行为是由接收处理器来控制的，它决定了事件的路由方式。

有一个默认的接收处理器，它包含了单个接收器，如果有接收器不属于任何接收器组，那么就会使用它。第2章中介绍的Hello World示例就使用了默认的接收处理器。单个接收器是不需要特别配置的。

为了让Flume能够知道接收器组，Flume提供了一个名为sinkgroups的新的顶层代理属性。类似于源、通道与接收器，你需要像下面这样为该属性加上代理名前缀：

```
agent.sinkgroups=sg1
```

这里我们为名为agent的代理定义了一个名为sg1的接收器组。

对于每个具名的接收器组，你需要通过sinks属性（包含了空格分隔的接收器名列表）指定它所包含的接收器：

```
agent.sinkgroups.sg1.sinks=k1,k2
```

上面表示为名为agent的代理定义了名为sg1的接收器组，其中包含了接收器k1与k2。

通常情况下，接收器组会与层次化的数据移动联合使用来对失

败进行路由。不过，它们还可用于写入到不同的 Hadoop 集群中，因为即便维护良好的集群也需要定期进行维护。

4.4.1　负载均衡

继续之前的示例，假如你想要均衡地对 k1 与 k2 的流量进行负载均衡。你需要指定一些额外的属性，如下表所示：

键	类型	默认值
processor.type	String	load_balance
processor.selector	String（round_robin，random）	round_robin
processor.backoff	boolean	false

如果将 processor.type 设为 load_balance，那就会使用循环选择，除非指定了 processor.selector 属性。你可以将该属性设为 round_robin 或是 random，还可以指定自己的负载均衡选择器机制，不过这里不会对其进行介绍。如果需要自定义控制，请参考 Flume 文档。

processor.backoff 属性指定如果重试了某个抛出异常的接收器时是否要使用异常备份。其默认值为 false，这意味着在抛出异常后下一次会重试，其顺序基于循环选择或是随机选择。如果设为 true，那么对于每次失败来说，等待时间将会翻倍，从一秒钟到大约 18 小时（2^{16} 秒）的上限。

在本书撰写之际，代码中 processor.backoff 的默认值为 false，不过 Flume 文档中写的却是 true。为了省却一些麻烦，请显式指定其值而不要使用默认值。

4.4.2　故障恢复

在使用某个接收器时，如果其不可用，那么你希望能够使用其

他的接收器，要想实现这样的效果，你需要将 processor.type 设为 failover。接下来，还需要设定其他一些属性来指定顺序，这是通过设置 processor.priority 属性并且后跟接收器名来实现的：

键	类型	默认值
processor.type	String	failover
processor.priority.NAME	int	
processor.maxpenality	int（毫秒）	30000

来看看下面这个示例：

```
agent.sinkgroups.sg1.sinks=k1,k2,k3
agent.sinkgroups.sg1.processor.type=failover
agent.sinkgroups.sg1.processor.priority.k1=10
agent.sinkgroups.sg1.processor.priority.k2=20
agent.sinkgroups.sg1.processor.priority.k3=20
```

优先级数字低的表示会优先使用，数字相同的则会随机使用。你可以随意使用任何编号方式（比如每次加 1、加 5 或是加 10 都可以）。在该示例中，会首先尝试使用接收器 k1，如果出现异常，那么接下来会尝试 k2 或 k3。如果先选择了 k3 并且不可用，那么还会尝试使用 k2。如果该接收器组中的所有接收器都不可用，那么该通道的事务就会回滚。

最后，processor.maxPenality 为组中不可用的接收器设定了一个指数退避上限。首次失败后要间隔一秒钟才可以再次使用。之后的失败都会将等待时间加倍，直到到达 processor.maxPenality 值为止。

4.5 小结

本章深入介绍了 HDFS 接收器，即会将数据流写到 HDFS 的 Flume 输出，介绍了 Flume 如何根据时间或是 Flume 头的内容将数据划分到不同的 HDFS 路径的方式。此外，本章还介绍了几种文件转储技

术，主要有：

- 基于时间的转储
- 基于事件数的转储
- 基于大小的转储
- 只在空闲时转储

我们还介绍了压缩技术，它主要用于降低 HDFS 的存储需求，在可能的情况下要尽量使用压缩。除了能够降低存储外，通常情况下读取一个压缩文件并在内存中对其进行解压缩要比读取未压缩的文件更快一些。这会提升使用该数据的 MapReduce Job 的性能。我们还介绍了压缩数据的分割，在确定使用哪种压缩算法时，这是一个决定因素。

接下来介绍了事件序列化器，可以通过它将 Flume 事件转化为外部存储格式，主要的事件序列化器有：

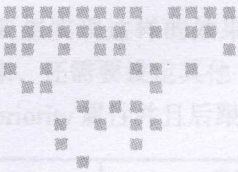
- 文本（只包含事件体）
- 带有头信息的文本（包含事件头与体）
- Avro 序列化（带有可选的压缩）

接下来介绍了各种文件格式，主要有：

- 序列文件（Hadoop 键值对文件）
- 数据流（未压缩的数据文件，比如 Avro 容器）
- 压缩的数据流

最后介绍了接收器组，接收器组可以通过负载均衡或是故障恢复等手段将事件路由到不同的源上，这样可以确保将数据路由到目的地时消除单点故障。

下一章将会介绍各种输入机制（源），这正是我们在第 3 章中配置的通道所接收的东西。



Chapter 5 第 5 章

源与通道选择器

既然已经介绍过通道与接收器，下面将会介绍将数据填充到 Flume 代理中的一些常见方式。正如在本书第 1 章中所介绍的那样，源（source）指的是 Flume 代理的输入点。Flume 发布包中提供了很多可用的源，此外还有众多的开源方案可供选择。就像大多数开源软件一样，如果找不到所需的源，那么你可以通过扩展 org.apache.flume.source.AbstractSource 类来编写自己的源。由于本书主要介绍的是如何将日志文件导入 Hadoop 中，因此我们将会介绍一些适合于此的源。

5.1 使用 tail 的问题

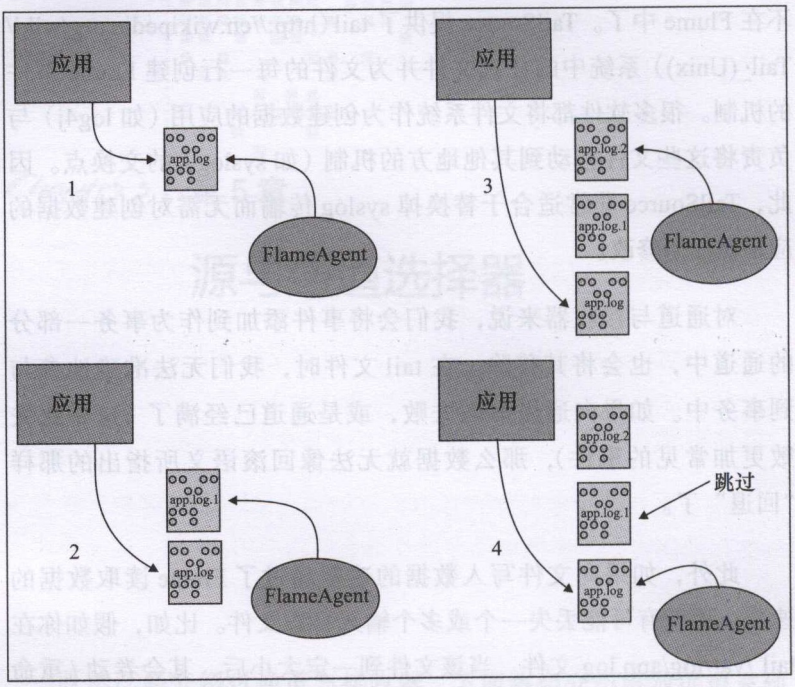
如果你使用过 Flume 0.9 系列版本，就会发现 TailSource 已经

不在 Flume 中了。TailSource 提供了 tail（http://en.wikipedia.org/wiki/Tail_(Unix)）系统中的任意文件并为文件的每一行创建 Flume 事件的机制。很多软件都将文件系统作为创建数据的应用（如 log4j）与负责将这些文件移动到其他地方的机制（如 syslog）的交换点。因此，TailSource 非常适合于替换掉 syslog 传输而无需对创建数据的应用做任何修改。

对通道与接收器来说，我们会将事件添加到作为事务一部分的通道中，也会将其移除。在 tail 文件时，我们无法准确地参与到事务中。如果向通道写入失败，或是通道已经满了（这是比失败更加常见的事件），那么数据就无法像回滚语义所指出的那样“回退”了。

此外，如果向文件写入数据的速度超过了 Flume 读取数据的速度，那就有可能丢失一个或多个输入日志文件。比如，假如你在 tail /var/log/app.log 文件。当该文件到一定大小后，其会卷动 / 重命名为 /var/log/app.log.1，并且新文件会从 /var/log/app.log 开始。假如某篇新闻的访问量增加，应用的日志比平时大了很多。当另一个卷动出现时，从 /var/log/app.log 移动到 /var/log/app.log.1，Flume 可能会读取卷动后的文件（/var/log/app.log.1）。现在 Flume 所读取的文件已经被重命名为 /var/log/app.log.2。当 Flume 读取该文件后，它会继续读取自己认为的下一个文件 /var/log/app.log，这样就会跳过 /var/log/app.log.1。这种数据丢失完全不会引起人们的注意，要尽量避免这种情况的发生。

基于这些原因，Flume 在重构时移除了 tail 功能。对移除的 TailSource 来说有一些解决办法，不过要注意的是，没有任何一种方式能够消除这些情况下数据丢失的可能。



5.2 exec 源

exec 源提供了在 Flume 外执行命令，然后将输出转换为 Flume 事件的机制。要想使用 exec 源，请将 type 属性设为 exec：

```
agent.sources.s1.type=exec
```

Flume 中的所有源都需要指定通道列表来写入事件，这是通过 channels 属性实现的。该列表可以接收一个或多个通道名，中间用空格分开：

```
agent.sources.s1.channels=c1
```

还有一个参数是必需的，那就是 command 属性，它告诉 Flume 传递给操作系统什么命令。例如：

```
agent.sources=s1
agent.sources.s1.channels=c1
agent.sources.s1.type=exec
agent.sources.s1.command=tail -F /var/log/app.log
```

这里为名为 agent 的代理配置了一个名为 s1 的源。该 exec 源会对 /var/log/app.log 文件执行 tail 命令，并且会追踪外部应用可能会对该日志文件所进行的任何卷动。所有的事件都会被写到 c1 通道。这是对 Flume 1.x 中移除掉 TailSource 的一种解决方案。

如果将 tail-F 命令与 exec 源一起使用，那么在 Flume 代理关闭或重启时，派生出来的进程不会保证 100% 关闭。这会导致出现永远不会退出的孤立的 tail 进程。根据定义，tail -F 是没有结束的。即便删掉了被 tail 的文件（在 Linux 中），运行中的 tail 进程也会一直开启该文件句柄。这会导致该文件空间不会被回收，直到该 tail 进程退出为止，然而该进程又不会退出。我想你现在应该知道为何 Flume 开发者不喜欢 tail 文件了吧。

如果沿着这条路继续走下去，请确保周期性地扫描父 PID 为 1 的 tail-F 进程表。它们实际上是无用进程，需要手工杀掉。

如下列表展示了可以与 exec 源搭配使用的一些属性：

键	是否必需	类型	默认值
type	是	String	exec
channels	是	String	空格分隔的通道列表
command	是	String	

（续）

键	是否必需	类型	默认值
restart	否	boolean	false
restartThrottle	否	long（毫秒）	10000毫秒
logStdErr	否	boolean	false
batchSize	否	int	20

并非每个命令都会一直保持运行状态（比如写入通道已经满了），也不是每个命令都会立刻退出。在该示例中，我们希望借助Linux uptime命令记录系统负载，它会向标准输出打印出一些系统信息然后退出：

```
agent.sources.s1.command=uptime
```

该命令会立刻退出，因此可以通过restart与restartThrottle属性周期性地运行它：

```
agent.sources.s1.command=uptime
agent.sources.s1.restart=true
agent.sources.s1.restartThrottle=60000
```

每隔一分钟会产生一个事件。在之前的tail示例中，如果由于通道填充问题导致exec源失败，那么你可以通过这些属性来重启exec源。在该示例中，设置restart属性会从当前文件的起始处开始tail文件，这会导致重复的情况发生。根据所设置的restartThrottle值的不同，你可能会因Flume外的文件卷动而丢失一些数据。此外，通道可能依然无法接收数据，这样源会再次失败。将该值设置得过低意味着留给通道释放数据的时间会变少，与我们所见到的一些接收器不同的是，它是没有指数退避选项的。

有时，命令会将你想要捕获的输出写到标准错误中。如果需要这些信息，那么请将logStdErr属性设为true。没有属性可以关闭掉

StdOut(不过可以将其过滤掉，方式是使用第 6 章中所介绍的拦截器)。

最后，可以通过修改 batchSize 属性来指定每个事务中要写入的事件数量。该属性的默认值是 20，如果输入数据过大，并且发现无法足够快地向通道中写入，那就需要将其值设得大一些。使用更大的 batchSize 值会降低每个事件的总体平均事务成本。请使用不同的值进行测试，并监控通道的写入率，这是唯一可靠的办法。

5.3　假脱机目录源

为了避免 tail 文件时所固有的那些问题，Flume 新增了一个源来追踪哪些文件被转换成了 Flume 事件，哪些文件尚待处理。假脱机目录源指的是一个日志满了再创建新日志。它认为复制到该目录中的文件是经过处理的；否则，源可能就是发送了部分文件。它还认为文件名永远不会发生变化；否则当重启后源就不知道发送了哪些文件，哪些文件还没有发送。其文件名的情况可以通过 log4j 中的 DailyRollingFileAppender 而非 RollingFileAppender 实现；不过，当前打开的文件需要写入到一个目录中并在关闭后复制到假脱机目录中。Log4J 则没有哪个追加器可以做到这一点。

也就是说，如果在你的环境中使用 Linux logrotate 程序将会很有趣。你可以通过 postrotate 脚本将处理完毕的文件移动到单独的目录中。

请记住，在 Flume 将文件标记为已发送后，你需要一个单独的进程来清理假脱机目录中的过期文件，否则磁盘最终将会被填满。

要想创建假脱机目录源，请将 type 属性设为 spooldir。必须将 spoolDir 属性设为待监控的目录：

```
agent.sources=s1
agent.sources.channels=c1
agent.sources.s1.type=spooldir
agent.sources.s1.spoolDir=/path/to/files
```

下表列出了假脱机目录源的属性：

键	是否必需	类型	默认值
type	是	String	spooldir
channels	是	String	空格分隔的通道列表
spoolDir	是	String	假脱机目录路径
fileSuffix	否	String	.COMPLETED
fileHeader	否	boolean	false
fileHeaderKey	否	String	file
batchSize	否	int	10
bufferMaxLines	否	int	100
maxBufferLineLength	否	int	5000

当文件传输完毕后，它会被重命名，如果没有设置 fileSuffix 属性，那么会加上一个 .COMPLETED 扩展。比如：

```
agent.source.s1.fileSuffix=.DONE
```

如果希望将绝对的文件路径附加到每个事件上，那么请将 fileHeader 属性设为 true。如果没有将 fileHeaderKey 属性设为其他值，那么这会使用 file 键创建一个头。比如：

```
agent.source.s1.fileHeader=true
agent.source.s1.fileHeaderKey=sourceFile
```

如果事件是从 /path/to/files/foo.1234.log 文件读取的，那么会添加头 {sourceFile=/path/to/files/foo.1234.log}。

可以通过 batchSize 属性调节每个事务中写到通道中的事件

数量。增加其值会提升吞吐量，代价则是更大的事务（还有可能是更大的回滚）。bufferMaxLines 属性用于设置读取文件时所用的内存缓冲大小，其大小是该值乘以 maxBufferLineLength 的结果。如果数据不大，那么可以考虑增加 bufferMaxLines 大小，降低 maxBufferLineLength 大小，这样会提升吞吐量而不会增加内存开销。也就是说，如果事件大小超过了 5000 个字符，那就需要将 maxBufferLineLength 设得更大一些。

最后，你需要确保无论使用哪种机制，将新文件写入假脱机目录中都会创建唯一的文件名，比如添加一个时间戳（可能还要添加其他一些信息）。重用文件名会干扰源，这样数据可能就得不到处理了。

跟往常一样，请记住，重启与错误会对假脱机目录中的文件创建重复的事件，这是因为没有被标记为已完成，文件会被重传。

5.4　syslog 源

syslog 已经有数十年的历史了，经常用作操作系统级的机制来捕获和移动系统日志。在很多方面，它与 Flume 提供的功能存在一些重叠。甚至还有一个名为 rsyslog 的 Hadoop 模块，它是 syslog 的一个更加现代化的变种（http://www.rsyslog.com/doc/rsyslog_conf_modules.html/omhdfs.html）。一般来说，我不喜欢版本之间没有任何关系的几项技术耦合在一起所形成的解决方案。如果使用 rsyslog/Hadoop 集成，那就需要更新 Hadoop 版本，同时还要将 Hadoop 集群更新到新的主版本上。在很大规模的服务器与环境下，这从逻辑上来说是很困难的。Hadoop 协议的向后兼容是在 Hadoop 社区中进行的，不过现在并非标准。我们

将在第7章中讨论分层数据流时对其进行深入介绍。

syslog拥有比较老的UDP传输和比较新的TCP协议，可以处理超过单个UDP包所能传输的数据量（大约64k），此外还能够处理网络相关的拥挤事件，这可能需要重新传输数据。

最后，一些syslog源的文档上并未提及的属性，可以为没有遵循RFC标准的消息提供额外的正则表达式匹配模式。这里并不会介绍这些额外的设置，不过如果频频遇到解析错误，那么你应该知道它们。如果出现这种情况，请参阅org.apache.flume.source.SyslogUtils的实现来寻找原因。

可以通过RFC 3164（http://tools.ietf.org/html/rfc3164）与RFC 5424（http://tools.ietf.org/html/rfc5424）来了解关于syslog术语（比如facility是什么）与标准格式的详细信息。

5.4.1 syslog UDP源

通常来说，在从服务器的本地syslog进程中获取数据时，syslog的UDP版本使用起来是安全的，前提是数据量要足够小（小于64k）。

无论网络的实际处理情况如何，该源的实现将2500字节作为最大的负载大小。因此，如果负载超过了这个大小，请使用TCP源。

要想创建syslog UDP源，请将type属性设为syslogudp。此外，必须将port属性设为监听端口。可选的host属性指定绑定地址。如果没有指定host，那么将会使用服务器所有可用的IP，这相当于指定0.0.0.0。在该示例中，我们只监听端口5140上的本地UDP连接：

```
agent.sources=s1
agent.sources.channels=c1
agent.sources.s1.type=syslogudp
agent.sources.s1.host=localhost
agent.sources.s1.port=5140
```

如果希望 syslog 能够转发 tail 的文件，那么可以像下面这样向 syslog 配置文件中添加一行代码：

```
*.err;*.alert;*.crit;*.emerg;kern.*      @localhost:5140
```

这会向 Flume 源发送所有级别的错误、警告、关键、紧急与内核消息。单个 @ 符号指定使用的是 UDP 协议。

下表展示了 syslog UDP 源的属性：

键	是否必需	类型	默认值
type	是	String	syslogudp
channels	是	String	空格分隔的通道列表
port	是	int	
host	否	String	0.0.0.0

syslog UDP 源所创建的 Flume 头信息总结如下：

头键	说明
Facility	syslog facility，参见 syslog 文档
Priority	syslog priority，参加 syslog 文档
timestamp	转换为纪元时间戳的 syslog 事件时间。如果无法从标准 RFC 格式进行转换则忽略
hostname	syslog 消息中解析后的主机名。如果无法解析则忽略
flume.syslog.status	解析 syslog 消息头时遇到了问题。如果负载没有遵循 RFC 则设为 Invalid。如果消息大小大于 eventSize 值（对于 UDP 来说内部将该值设为 2500）时设为 Incomplete。如果一切正常则忽略掉

5.4.2 syslog TCP 源

如前所述，syslog TCP 源为 TCP 之上传输的消息提供了端点，这考虑到了更大的负载与 TCP 重试语义，应该将其用在任何可靠的内部服务器通信上。

要想创建 syslog TCP 源，请将 type 属性设为 syslogtcp，还需要设定绑定地址与监听端口号：

```
agent.sources=s1
agent.sources.s1.type=syslogtcp
agent.sources.s1.host=0.0.0.0
agent.sources.s1.port=12345
```

如果 syslog 实现支持 TCP 之上的 syslog，那么配置与上面完全一样，只不过这里要使用两个 @ 符号来表示 TCP 传输。还是使用之前那个示例，只不过这里用的是 TCP，将信息转发给运行在 flume-1 服务器上的 Flume 代理：

```
*.err;*.alert;*.crit;*.emerg;kern.*      @@flume-1:12345
```

syslog TCP 源有几个可选属性：

键	是否必需	类型	默认值
type	是	String	syslogtcp
channels	是	String	空格分隔的通道列表
port	是	int	
host	否	String	0.0.0.0
eventSize	否	int（字节）	2500 字节

syslog TCP 源所创建的 Flume 头信息总结如下：

头键	说明
Facility	syslog facility，参见 syslog 文档
Priority	syslog priority，参加 syslog 文档
timestamp	转换为纪元时间戳的 syslog 事件时间。如果无法从标准 RFC 格式进行转换则忽略
hostname	syslog 消息中解析后的主机名。如果无法解析则忽略
flume.syslog.status	解析 syslog 消息头时遇到了问题。如果负载没有遵循 RFC 则设为 Invalid。如果消息大小大于配置的 eventSize 值时设为 Incomplete。如果一切正常则忽略

5.4.3　多端口 syslog TCP 源

多端口 syslog TCP 源在功能上与 syslog TCP 源几乎一样，只不过它可以监听多个端口的输入。如果无法在转发规则中修改 syslog 所用的端口则可以使用它（可能因为服务器不是你的）。此外，你可以通过它使用一个源来读取多种格式，然后写到不同的通道中。在 5.5 节中将会对其进行介绍。

要想配置该源，请将 type 属性设为 multiport_syslogtcp：

```
agent.sources.s1.type=multiport_syslogtcp
```

就像其他 syslog 源一样，你需要指定端口，不过这里要指定的是一个空格分隔的端口列表。只有在指定了一个端口的情况下才可以使用它。该属性是 ports（复数）：

```
agent.sources.s1.type=multiport_syslogtcp
agent.sources.s1.channels=c1
agent.sources.s1.ports=33333 44444
agent.sources.s1.host=0.0.0.0
```

上面配置了一个名为 s1 的多端口 syslog TCP 源，监听端口 33333 与 44444 上的连接并将其发送到通道 c1。

为了显示事件来自哪个端口，你可以将可选的portHeader属性设定为键的名字，而键的值则为端口号。如果将该属性添加到配置中：

```
agent.sources.s1.portHeader=port
```

接下来，来自于端口33333的任何事件都会有一个键值为{"port"="33333"}的头。正如第4章中所介绍的那样，现在可以将该值（实际上任何头都可以）作为HDFS接收器文件路径约定的一部分，如下所示：

```
agent.sinks.k1.hdfs.path=/logs/%{hostname}/%{port}/%Y/%m/%D/%H
```

下面是完整的属性列表：

键	是否必需	类型	默认值
type	是	String	syslogtcp
channels	是	String	空格分隔的通道列表
ports	是	int	空格分隔的端口号列表
host	否	String	0.0.0.0
eventSize	否	int（字节）	2500字节
portHeader	否	String	
batchSize	否	int	100
readBufferSize	否	int（字节）	1024
numProcessors	否	int	自动检测
charset.default	否	String	UTF-8
charset.port.PORT#	否	String	

相比于标准的TCP syslog源，该TCP源有几个额外的可调选项，有时你可能会用到。首先是batchSize属性，它指的是通道每个事务所处理的事件数量。readBufferSize属性则指定了内部的Mina库所用的内部缓冲大小。最后，numProcessors属性用于设定Mina中的工作线程池大小。在开始调节这些参数前要先熟悉一下

Mina（http://mina.apache.org/），如果不使用默认值，那么请先看看源代码。

最后，在进行字符串与字节数组之间的转换时，你可以指定默认的字符编码，也可以针对每个端口号来指定。

```
agent.sources.s1.charset.default=UTF-16
agent.sources.s1.charset.port.33333=UTF-8
```

上述示例配置表明所有的端口号都将使用 UTF-16 编码进行解释，而端口 33333 则会使用 UTF-8。

该源所创建的 Flume 头信息总结如下：

头键	说明
Facility	syslog facility，参见 syslog 文档
Priority	syslog priority，参加 syslog 文档
timestamp	转换为纪元时间戳的 syslog 事件时间。如果无法从标准 RFC 格式进行转换则忽略
hostname	syslog 消息中解析后的主机名。如果无法解析则忽略
flume.syslog.status	解析 syslog 消息头时遇到了问题。如果负载没有遵循 RFC 则设为 Invalid。如果消息大小大于配置的 eventSize 值时设为 Incomplete。如果一切正常则忽略

5.5　通道选择器

正如在第 1 章所介绍的那样，一个源可以写到一个或多个通道中。这也正是属性为复数的原因（属性是 channels 而非 channel）。处理多个通道的方式有两种。事件可以写到所有通道中，或根据某个 Flume 头值而仅写到一个通道中。Flume 的这种内部机制叫作通道选择器。

可以通过selector.type属性指定任意通道的选择器。任何特定于选择器的属性都以常规的源前缀开头，后跟代理名字、关键字sources以及源名字，如下所示：

```
agent.sources.s1.selector.type=replicating
```

5.5.1 复制

在默认情况下，如果没有为源指定选择器，那么其默认值为replicating。replicating选择器会将相同的事件写到源的通道列表的所有通道中：

```
agent.sources.s1.channels=c1 c2 c3
agent.sources.s1.selector.type=replicating
```

在该示例中，每个事件都会被写到所有3个通道中，即c1、c2与c3。

该选择器还有一个名为optional的可选属性。它是个可选的空格分隔的通道列表，设定代码如下：

```
agent.sources.s1.channels=c1 c2 c3
agent.sources.s1.selector.type=replicating
agent.sources.s1.selector.optional=c2 c3
```

写入到通道c2或c3时所遇到的任何失败都不会造成事务失败，写入到c1的任何数据都会被提交。在之前没有可选通道的示例中，任何一个通道失败都会导致整个事务回滚。

5.5.2 多路复用

如果想向不同的通道发送不同的事件，那就应该使用多路复用

通道选择器，方式是将 selector.type 设为 multiplexing。还需要告诉通道选择器使用哪个头，方式是设置 selector.header 属性。

```
agent.sources.s1.selector.type=multiplexing
agent.sources.s1.selector.header=port
```

假设我们使用多端口 syslog TCP 源监听 4 个端口 11111、22222、33333 与 44444，并且将 portHeader 设为 port。考虑如下配置：

```
agent.sources.s1.selector.default=c2
agent.sources.s1.selector.mapping.11111=c1 c2
agent.sources.s1.selector.mapping.44444=c2
agent.sources.s1.selector.optional.44444=c3
```

这会导致端口 22222 与 33333 的流量只会进入通道 c2 中。端口 11111 的流量会进入通道 c1 与 c2 中。无论哪个通道失败都会导致这两个通道接收不到任何东西。端口 44444 的流量会进入通道 c2 与 c3 中；不过，如果写入 c3 失败则不影响事务向通道 c2 提交（并且 c3 将不会再次尝试该事件）。

5.6　小结

本章深入介绍了各种源，你可以使用它们将日志数据纳入 Flume 中，这包括：

- exec 源
- syslog 源（UDP、TCP 与多端口 TCP）

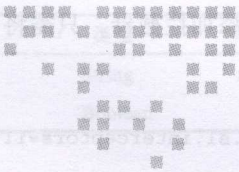
本章介绍了 Flume 0.9 中过时的 TailSource 的功能以及使用 tail 语义的问题。

本章还介绍了通道选择器以及如何将事件发送到一个或多个通道中。特别是：

- 复制通道选择器
- 多路复用通道选择器

此外，本章还介绍了可选通道，通过它可以实现单一通道的失败；如果有多个通道可用，还可以将事务只应用到其中几个通道上。

下一章将会介绍拦截器，它可以实现即时检测与事件转换。如果搭配上通道选择器，拦截器则提供了创建 Flume 复杂数据流的终极武器。



第 6 章 Chapter 6

拦截器、ETL 与路由

数据处理管道中所必需的最后一个功能就是即时检测与转换事件的能力，这可以通过拦截器实现。正如第 1 章所介绍的，拦截器可以插入源之后或接收器之前。

6.1 拦截器

拦截器的功能可以通过下面这种方法加以概括：

```
public Event intercept(Event event);
```

它作为 Flume 事件传递，并且以 Flume 事件的形式返回。它可以什么都不做，也就是说返回相同的事件。通常情况下，它会以某种方式修改事件。如果返回 null，则表明事件被丢弃了。

要想向源添加拦截器，只需将 interceptors 属性添加到源的名字中即可。比如：

```
agent.sources.s1.interceptors=i1 i2 i3
```

这为名为 agent 的代理的源 s1 定义了 3 个拦截器：i1、i2 与 i3。

拦截器会按照列出的顺序运行。在上述示例中，i2 会接收到来自 i1 的输出，i3 会接收到来自 i2 的输出；通道选择器会接收到来自 i3 的输出。

既然已经根据名字定义好了拦截器，接下来需要指定其类型，如下所示：

```
agent.sources.s1.interceptors.i1.type=TYPE1
agent.sources.s1.interceptors.i1.additionalProperty1=VALUE
agent.sources.s1.interceptors.i2.type=TYPE2
agent.sources.s1.interceptors.i3.type=TYPE3
```

下面来看看 Flume 自带的一些拦截器，从而更好地了解如何对其进行配置。

6.1.1 Timestamp

正如其名字所示，如果不存在，Timestamp 拦截器会向 Flume 事件添加一个键为 timestamp 的头。要想使用它，请将 type 属性设为 timestamp。

如果事件已经包含了 timestamp 头，那么它会被当前的时间覆盖，除非通过将 preserveExisting 属性设为 true 来保留之前的值。

下表列出了 timestamp 拦截器的属性：

键	是否必需	类型	默认值
type	是	String	timestamp
preserveExisting	否	Boolean	false

如果只想向源中添加一个 timestamp 头（之前不存在），下面展示的则是一个完整的配置：

```
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.i1.type=timestamp
agent.sources.s1.interceptors.i1.preserveExisting=true
```

回忆一下第 4 章所介绍的 HDFSSink 路径，我们使用了事件日期：

```
agent.sinks.k1.hdfs.path=/logs/apache/%Y/%m/%D/%H
```

timestamp 头确定了这个路径。如果没有，那么 Flume 不知道在哪里创建文件，你也无法获得相应的结果。

6.1.2　Host

从简单性上来说，Host 拦截器类似于 Timestamp 拦截器，它会向事件添加一个包含当前 Flume 代理 IP 地址的头。要想使用 Host 拦截器，请将 type 属性设为 host。

```
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.type=host
```

如果没有指定 hostHeader 属性，那么该头的键就是 host。与之前类似，现有的头将会被覆盖，除非将 preserveExisting 属性设为 true。最后，如果希望值是主机名的反向 DNS 而非 IP，那么请将 useIP 属性设为 false。记住，反向 DNS 查找会增加数据流的处理时间。

下表列出了 Host 拦截器的属性：

键	是否必需	类型	默认值
type	是	String	host
hostHeader	否	String	host
preserveExisting	否	Boolean	false
useIP	否	Boolean	true

如果只想为每个事件添加包含代理的 DNS 主机名的 relayHost 头，下面展示的则是一个完整的配置：

```
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.i1.type=host
agent.sources.s1.interceptors.i1.hostHeader=relayHost
agent.sources.s1.interceptors.i1.useIP=false
```

如果希望将事件在数据流中所经过的路径记录下来，那么该拦截器就会很有用。然而，很多时候你对事件的起源而非其所经过的路径更感兴趣，这也是我还未使用该拦截器的原因所在。

6.1.3 Static

Static 拦截器用于向每个处理的 Flume 事件插入任意单个的键 / 值头。如果需要多个键 / 值，添加多个 Static 拦截器即可。与之前介绍的拦截器不同的是，其默认行为是保留已存在的具有相同键的头。我的建议是总是指定你所需的而不要依赖于默认值。

我很奇怪的是既然默认值没什么用，那为何键值属性不是必需的呢？

下表列出了 Static 拦截器的属性：

键	是否必需	类型	默认值
type	是	String	static
key	否	String	key
value	否	String	value
preserveExisting	否	Boolean	true

最后来看一个示例配置，该配置向事件中插入了两个之前不存在的头：

```
agent.sources.s1.interceptors=pos env
agent.sources.s1.interceptors.pos.type=static
agent.sources.s1.interceptors.pos.key=pointOfSale
agent.sources.s1.interceptors.pos.value=US
agent.sources.s1.interceptors.env.type=static
agent.sources.s1.interceptors.env.key=environment
agent.sources.s1.interceptors.env.value=staging
```

6.1.4　正则表达式过滤

如果想根据体的内容来过滤事件，那么正则表达式过滤拦截器就是你所需要的。根据所提供的正则表达式，它要么过滤掉匹配的事件，要么仅保留匹配的事件。首先要将拦截器的 type 属性设为 regex_filter。你希望匹配的模式是通过 Java 风格的正则表达式语法来指定的。请参见 JavaDoc 来了解使用细节：

```
http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html.
```

模式字符串是通过 regex 属性设定的。最后，你需要告诉拦截器是否想要排除匹配的记录，这是通过将 excludeEvents 属性设为 true 实现的。其默认值（false）表示只保留与模式匹配的事件。

下表列出了正则表达式过滤拦截器的属性：

键	是否必需	类型	默认值
type	是	String	regex_filter
regex	否	String	.*
excludeEvents	否	Boolean	false

在下面这个示例中，包含字符串 NullPointerException 的任何事件都会被丢弃：

```
agent.sources.s1.interceptors=npe
agent.sources.s1.interceptors.npe.type=regex_filter
agent.sources.s1.interceptors.npe.regex=NullPointerException
agent.sources.s1.interceptors.npe.excludeEvents=true
```

6.1.5 正则表达式抽取

有时，你想要将事件体的内容抽取出来并放到 Flume 头中以便可以通过通道选择器执行路由。可以使用正则表达式抽取过滤器来实现该功能。首先要将拦截器的 type 属性设为 regex_extractor。

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
```

类似于正则表达式过滤拦截器，正则表达式抽取使用了 Java 风格的正则表达式语法。为了抽取一个或多个字段，首先通过分组匹配圆括号设定 regex 属性。假设我们要根据“Error: N”形式寻找事件中的错误号，其中 N 表示数字：

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=Error:\\s(\\d+)
```

如你所见，我用捕获圆括号围住了数字，它可能包含一个或多个数字。既然已经匹配了所需的模式，接下来就需要告诉 Flume 该

如何处理匹配结果。这里需要引入序列化器，它提供了一种可插拔的机制来解释每一个匹配。在该示例中，我只得到了一个匹配，因此空格分隔的序列化器名字列表中只有一个条目：

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=Error:\\s(\\d+)
agent.sources.s1.interceptors.e1.serializers=ser1
agent.sources.s1.interceptors.e1.serializers.ser1.type=default
agent.sources.s1.interceptors.e1.serializers.ser1.name=error_no
```

name属性指定了要使用的事件的键，其值是正则表达式的匹配文本。default这个type值（如果没有指定也是其默认值）是个简单的穿越序列化器。对于如下事件体：

```
NullPointerException: A problem occurred. Error: 123. TxnID: 5X2T9E.
```

下面这个头将会被添加到事件中：

```
{ "error_no":"123" }
```

如果想要添加TxnID值作为头，只需再添加一个匹配模式分组与序列化器即可：

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=Error:\\s(\\d+).*TxnID:\\s(\\w+)
agent.sources.s1.interceptors.e1.serializers=ser1 ser2
agent.sources.s1.interceptors.e1.serializers.ser1.type=default
agent.sources.s1.interceptors.e1.serializers.ser1.name=error_no
agent.sources.s1.interceptors.e1.serializers.ser2.type=default
agent.sources.s1.interceptors.e1.serializers.ser2.name=txnid
```

对上述输入来说，这会创建如下的头：

```
{ "error_no":"123", "txnid":"5x2T9E" }
```

不过，如果将字段翻转过来，像下面这样：

```
NullPointerException: A problem occurred. TxnID: 5X2T9E. Error: 123.
```

那么结果将只有一个针对TxnID的头。处理这种排序的更好的方式是使用多个拦截器，这样顺序就不重要了：

```
agent.sources.s1.interceptors=e1 e2
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=Error:\\s(\\d+)
agent.sources.s1.interceptors.e1.serializers=ser1
agent.sources.s1.interceptors.e1.serializers.ser1.type=default
agent.sources.s1.interceptors.e1.serializers.ser1.name=error_no
agent.sources.s1.interceptors.e2.type=regex_extractor
agent.sources.s1.interceptors.e2.regex=TxnID:\\s(\\w+)
agent.sources.s1.interceptors.e2.serializers=ser1
agent.sources.s1.interceptors.e2.serializers.ser1.type=default
agent.sources.s1.interceptors.e2.serializers.ser1.name=txnid
```

除了穿透序列化器外，Flume自带的另外一个序列化器实现要通过类的全名org.apache.flume.interceptor.RegexExtractorInterceptor-MillisSerializer指定。该序列化器用于将时间转换为毫秒数。你需要基于org.joda.time.format.DateTimeFormat模式来指定pattern属性。

假设你要接收Apache Web服务器访问日志：

```
192.168.1.42 - - [29/Mar/2013:15:27:09 -0600] "GET /index.html
HTTP/1.1" 200 1037
```

其完整的正则表达式如下所示（以Java字符串形式给出，其中的反斜杠与引号通过额外的反斜杠进行转义）：

```
^([\\d.]+) \\S+ \\S+ \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.+?)\"
(\\d{3}) (\\d+)
```

与org.joda.time.format.DateTimeFormat模式匹配的时间模式如

下所示：

```
yyyy/MMM/dd:HH:mm:ss Z
```

这样，我们的配置就如下代码所示：

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=^([\\d.]+) \\S+ \\S+ \\
[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.+?)\" (\\d{3}) (\\d+)
agent.sources.s1.interceptors.e1.serializers=ip dt url sc bc
agent.sources.s1.interceptors.e1.serializers.ip.name=ip_address
agent.sources.s1.interceptors.e1.serializers.dt.type=org.apache.flume.
interceptor.RegexExtractorInterceptorMillisSerializer
agent.sources.s1.interceptors.e1.serializers.dt.pattern=yyyy/MMM/
dd:HH:mm:ss Z
agent.sources.s1.interceptors.e1.serializers.dt.name=timestamp
agent.sources.s1.interceptors.e1.serializers.url.name=http_request
agent.sources.s1.interceptors.e1.serializers.sc.name=status_code
agent.sources.s1.interceptors.e1.serializers.bc.name=bytes_xfered
```

对之前的示例来说，会创建如下的头：

```
{ "ip_address":"192.168.1.42", "timestamp":"1364588829",
"http_request":"GET /index.html HTTP/1.1", "status_code":"200",
"bytes_xfered":"1037" }
```

体的内容是不受影响的。此外，你会发现我没有为其他的序列化器的 type 属性指定 default，因为它就是默认值。

该拦截器没有覆写检查。比如，事件之前存在时间值，那么使用 timestamp 键会将其覆盖掉。

你可以通过实现 org.apache.flume.interceptor.RegexExtractorInterceptorSerializer 接口为该拦截器实现自己的序列化器。然而，如果目标是将事件体的数据移动到头中，那么你只需要实现一个自定义

拦截器即可，这样除了设置头的值外，还可以修改体的内容，否则数据将会重复。

下表列出了该拦截器的属性：

键	是否必需	类型	默认值
type	是	String	regex_extractor
regex	是	String	
serializers	是	空格分隔的序列化器名字列表	
serializers.NAME.name	是	String	
serializers.NAME.type	否	默认或实现的完整名字	default
serializers.NAME.PROP	否	特定于序列化器的属性	

6.1.6 自定义拦截器

如果想将一些自定义代码添加到Flume实现中，那么你需要的就是自定义拦截器。如前所述，你需要实现org.apache.flume.interceptor.Interceptor接口以及与之相关的org.apache.flume.interceptor.Interceptor.Builder接口。

假如需要对事件体进行URL解码，代码如下所示：

```
public class URLDecode implements Interceptor {


  public void initialize() {}

  public Event intercept(Event event) {
    try {
      byte[] decoded = URLDecoder.decode(new String(event.getBody()),
"UTF-8").getBytes("UTF-8");
      event.setBody(decoded);
    } catch UnsupportedEncodingException e) {
```

```
    // This shouldn't happen. Fall through to unaltered event.
    }
    return event;
  }

  public List<Event> intercept(List<Event> events) {
    for (Event event:events) {
      intercept(event);
    }
    return events;
  }

  public void close() {}

  public static class Builder implements Interceptor.Builder {
    public Interceptor build() {
      return new URLDecode();
    }
    public void configure(Context context) {}
  }
}
```

要想配置新的拦截器，请将type属性设为Build类的完全限定名：

```
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.i1.type=com.example.URLDecoder$Builder
```

要想了解关于如何传递并验证属性的更多示例，请查看Flume现有的拦截器实现源代码。

请记住，自定义拦截器中的任何耗时耗空间的处理都会对整体的吞吐量造成影响，因此请留意实现中的对象群或是计算密集型处理。

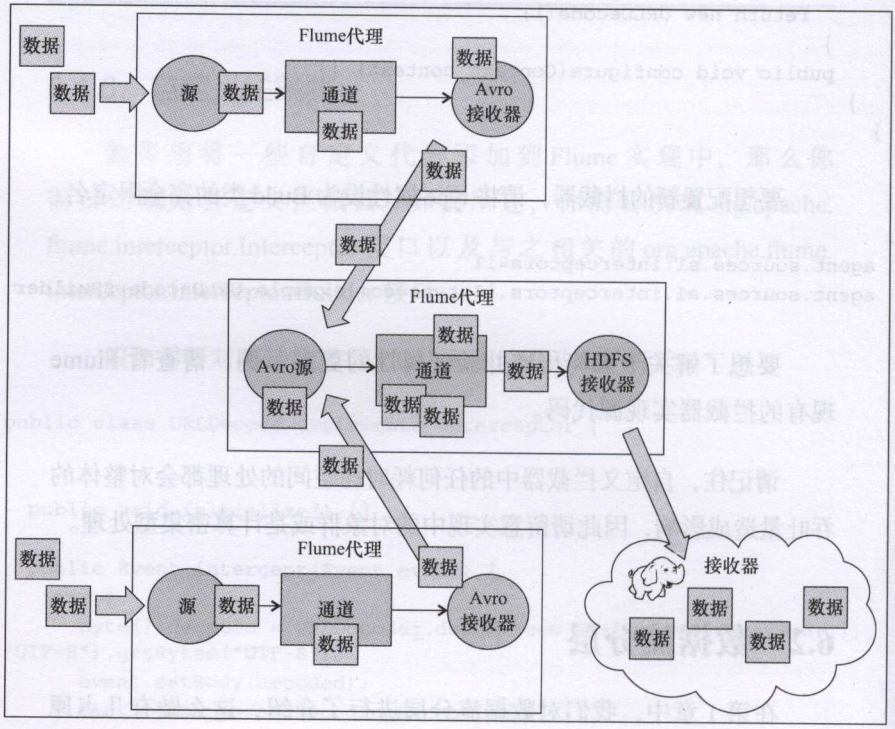
6.2 数据流分层

在第1章中，我们对数据流分层进行了介绍，这么做有几点原

因。你可能想要限制直接连接到Hadoop集群的Flume代理的数量，从而限制并行请求的数量；还可能是因为在维护Hadoop集群时应用服务器上缺少足够的磁盘空间来存储大量的数据。无论什么原因，什么情况，将Flume代理链接起来的最常见机制还是使用Avro源/接收器对。

6.2.1 Avro源/接收器

第4章对Avro做过一些介绍，当时提到将其作为存储在HDFS中的文件的磁盘序列化格式。这里将使用它作为Flume代理间的通信。典型的配置如下所示：



要想使用 Avro 源，请将 type 属性值设为 avro。你需要提供一个绑定地址与监听端口号：

```
collector.sources=av1
collector.sources.av1.type=avro
collector.sources.av1.bind=0.0.0.0
collector.sources.av1.port=42424
collector.sources.av1.channels=ch1
collector.channels=ch1
collector.channels.ch1.type=memory
collector.sinks=k1
collector.sinks.k1.type=hdfs
collector.sinks.k1.channel=ch1
collector.sinks.k1.hdfs.path=/path/in/hdfs
```

这里我们将右侧的代理配置为监听端口 42424、使用 memory 通道，并且写到 HDFS 中。使用 memory 通道的目的在于简化这个示例配置。此外，注意到我为这个代理起了个不同的名字 collector 以防止产生混乱。

左侧的代理用于向 collector 层提供数据，其配置是类似的。出于简化的目的，这里省略了该配置的源：

```
client.channels=ch1
client.channels.ch1.type=memory
client.sinks=k1
client.sinks.k1.type=avro
client.sinks.k1.channel=ch1
client.sinks.k1.hostname=collector.example.com
client.sinks.k1.port=42424
```

hostname 值 collector.example.com 与该机器上的代理名没有任何关系，它是接收 Avro 源的目标机器的主机名（也可以使用 IP）。该配置（名为 client）可以应用到左侧的两个代理上，前提是假设这两个代理有类似的源配置。

我很不喜欢单点故障，因此使用上述配置信息配置了两个collector代理，并且对使用一个接收器组的两个客户端代理采用了循环方式来使用。另外，出于简化的目的，这里省略了源。

```
client.channels=ch1
client.channels.ch1.type=memory
client.sinks=k1 k2
client.sinks.k1.type=avro
client.sinks.k1.channel=ch1
client.sinks.k1.hostname=collectorA.example.com
client.sinks.k1.port=42424
client.sinks.k2.type=avro
client.sinks.k2.channel=ch1
client.sinks.k2.hostname=collectorB.example.com
client.sinks.k2.port=42424
client.sinkgroups=g1
client.sinkgroups.g1=k1 k2
client.sinkgroups.g1.processor.type=load_balance
client.sinkgroups.g1.processor.selector=round_robin
client.sinkgroups.g1.processor.backoff=true
```

6.2.2 命令行Avro

Avro源还可以与本书第2章中介绍的命令行选项搭配使用。相对于使用agent参数运行flume-ng来说，你可以通过传递avro-client参数向Avro源发送一个或多个文件。从帮助文档中可以看到，这些是avro-client特有的选项：

```
avro-client options:
  --dirname <dir>          directory to stream to avro source
  --host,-H <host>         hostname to which events will be sent
(required)
  --port,-p <port>         port of the avro source (required)
  --filename,-F <file>  text file to stream to avro source [default:
std input]
  --headerFile,-R <file> headerFile containing headers as key/value
pairs on each new line
  --help,-h                display help text
```

这对于测试来说非常有帮助，可以在出错的情况下手动再次发送数据或是导入存储于其他地方的老数据。

就像Avro接收器一样，你需要指定发送数据的主机名与端口号。可以通过--filename选项发送单个文件，也可以通过--dirname选项将目录中的所有文件全部发送出去。如果这两个参数都未指定，那么默认将会使用stdin。如下代码展示了如何向之前配置的Flume代理发送名为foo.log的文件：

```
$ ./flume-ng avro-client --filename foo.log --host collector.example.
com --port 42424
```

每一行输入都会被转换为一个Flume事件。

你还可以指定一个包含键/值对的文件来设置Flume头值，该文件使用的是Java属性文件语法。如果有一个名为headers.properties的文件：

```
pointOfSale=US
environment=staging
```

那么使用--headerFile选项将会对每个创建的事件设定这两个头：

```
$ ./flume-ng avro-client --filename foo.log --headerFile headers.
properties --host collector.example.com --port 42424
```

6.2.3 Log4J追加器

正如第5章中所介绍的那样，使用文件系统文件作为源可能会产生问题。避免这个问题的一个手段是在Java应用中使用Flume Log4J追加器。在内部，它使用的是与Avro接收器所用的相同的

Avro 通信，因此你只需配置它向 Avro 源发送数据即可。

该追加器有两个属性，如下 XML 所示：

```
<appender name="FLUME" class="org.apache.flume.clients.log4jappender.
Log4jAppender">
  <param name="Hostname" value="collector.example.com"/>
  <param name="Port" value="42424"/>
</appender>
```

体的格式由追加器配置的布局（这里没有列出来）所指定。与 Flume 头对应的 Log4J 字段如下表所示：

Flume 头键	Log4J LoggingEvent 字段
flume.client.log4j.logger.name	event.getLoggerName()
flume.client.log4j.log.level	event.getLevel()，是个数字。参见 org.apache.log4j.Level
flume.client.log4j.timestamp	event.getTimeStamp()
flume.client.log4j.message.encoding	未定义，总是 UTF8
flume.client.log4j.logger.other	只在之前的字段映射出现问题时才会用到它，因此正常情况下是不会出现的

请参见 http://logging.apache.org/log4j/1.2/ 以了解关于 Log4J 用法的详细信息。

你需要在运行期将 flume-ng-sdk JAR 加入到 Java 应用的类路径中来使用 Flume 的 Log4J 追加器。

请记住，如果向 Avro 源发送数据时出现了问题，那么该追加器会抛出异常，并且日志消息会被丢弃，因为没有地方存储它。将其放到内存中会急速增加 JVM 堆的负担，这可是比丢弃数据记录更为糟糕的做法。

6.2.4　负载均衡 Log4J 追加器

你可能已经发现之前的 Log4J 追加器配置中只有一个主机名 / 端口号。如果希望将负载分布到多个 collector 代理上（出于更大容量或是容错的目的），你可以使用 LoadBalancingLog4jAppender。该追加器只有一个名为 Hosts 的必填属性，它是个空格分隔的主机名与端口号列表，其中主机名与端口号之间使用分号分隔：

```
<appender name="FLUME" class="org.apache.flume.clients.log4jappender.
LoadBalancingLog4jAppender">
  <param name="Hosts" value="server1:42424 server2:42424"/>
</appender>
```

还有个名为 Selector 的可选属性，它指定了负载均衡方法。有效值包括 RANDOM 和 ROUND_ROBIN。如果没有指定，那么其默认值是 RANDOM。你可以实现自己的选择器，不过这超出了本书的讨论范围。如果感兴趣，可以看看 LoadBalancingLog4jAppender 类的源代码。

如果没有指定，那么负载均衡 Log4J 追加器的默认选择器机制将是随机。你会发现这与第 4 章中所介绍的接收器组的类似功能不同，它的默认选择器值是 round robin。

这又一次表明你应该总是明确指定自己的意图而不要依赖于默认值。

最后，还有另一个可选属性用于在无法连接到服务器时覆盖最大的指数回退时间。一开始，如果无法连接到服务器，那么在重试前需要等待一秒钟。每次无法连接到服务器时，重试时间都会翻倍，一直到默认的 30 秒最大值。如果希望将该最大值增加到

2分钟，那就需要指定MaxBackoff属性（单位是毫秒），如下代码所示：

```
<appender name="FLUME" class="org.apache.flume.clients.log4jappender.
LoadBalancingLog4jAppender">
  <param name="Hosts" value="server1:42424 server2:42424"/>
  <param name="Selector" value="ROUND_ROBIN"/>
  <param name="MaxBackoff" value="120000"/>
</appender>
```

在该示例中，我们还覆盖了默认的随机选择器，使用了round robin。

6.3 路由

现在，你已经了解了Flume中的各种机制，因此理解如何根据内容将数据路由到不同目的地也将是很轻松的事情了。

如果头不存在，那么首先要通过源端的拦截器来得到想要纳入到Flume头中的数据。接下来，针对该头的值使用多路复用通道选择器将数据切换到另一个通道中。

假如你想要捕获所有进入到HDFS中的异常。在该配置下，你会看到事件通过端口42424上的Avro进入到了源s1。然后检测该事件，看看其体中是否包含了文本“Exception”。如果包含，那就会创建一个头键exception（其值为Exception）。该头用于将这些事件切换到通道c1，最后到达HDFS。如果事件与模式不匹配，那么它就不会有exception头，并且会通过默认选择器进入到通道c2中，然后通过Avro序列化器转发到服务器foo.example.com的12345端口上。

```
agent.sources=s1
agent.sources.s1.type=avro
agent.sources.s1.bind=0.0.0.0
agent.sources.s1.port=42424
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.i1.type=regex_extractor
agent.sources.s1.interceptors.i1.regex=(Exception)
agent.sources.s1.interceptors.i1.serializers=ex
agent.sources.s1.intercetpros.i1.serializers.ex.name=exception
agent.sources.s1.selector.type=multiplexing
agent.sources.s1.selector.header=exception
agent.sources.s1.selector.mapping.Exception=c1
agent.sources.s1.selector.default=c2
agent.channels=c1 c2
agent.channels.c1.type=memory
agent.channels.c2.type=memory
agent.sinks=k1 k2
agent.sinks.k1.type=hdfs
agent.sinks.k1.channel=c1
agent.sinks.k1.hdfs.path=/logs/exceptions/%y/%M/%d/%H
agent.sinks.k2.type=avro
agent.sinks.k2.channel=c2
agent.sinks.k2.hostname=foo.example.com
agent.sinks.k2.port=12345
```

6.4　小结

本章介绍了 Flume 自带的如下拦截器：

- Timestamp：用于添加一个时间戳头，可能会覆盖已有的值。
- Host：用于添加 Flume 代理主机名或 IP，作为事件的头。
- Static：用于添加静态的字符串头。
- 正则表达式过滤：用于根据匹配的正则表达式包含或是排除掉事件。
- 正则表达式抽取：用于从匹配的正则表达式头中创建头，对于通道选择器路由也是很有帮助的。

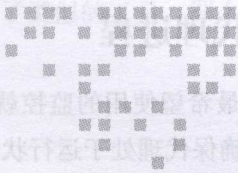
- 自定义：如果没有现成的，那么可以通过它创建自定义转换。

本章还介绍了如何通过 Avro 源与接收器对数据流进行分层。

接下来介绍了两个 Log4J 追加器，一个使用单一路径，另一个使用了负载均衡，可以直接与 Java 应用进行集成。

最后通过示例演示了如何联合使用拦截器与通道选择器实现路由决策逻辑。

下一章将会介绍如何通过 Ganglia 来监控 Flume 数据流。



第 7 章 Chapter 7

监控 Flume

Flume 的用户指南说道：

> Flume 的监控是个依旧在进行当中的工作，变化可能会经常发生。有几个 Flume 组件会将度量结果报告给 JMX 平台 MBean 服务器，可以通过 JConsole 查询这些度量结果。

虽然 JMX 对于一般的度量值的浏览来说很不错，不过如果有成百上千台服务器发送数据，那眼睛肯定是不够用了。你需要的是能够一下子查看一切的方式。不过什么是重要的东西呢？这是个难以回答的问题，我会试着在本章的监控选项中介绍我认为的几项重要指标。

7.1 监控代理进程

显然，你最希望使用的监控就是对Flume代理进行的监控了，也就是说，要确保代理处于运行状态。有很多产品可以实现这种进程监控，这里无法将其一一列举出来。无论你在多大规模的公司工作，肯定已经有做这件事的系统存在了。如果事实如此，那就不要再构建自己的了。运维所要的最后一个东西就是能够24×7显示监控情况的屏幕。

7.1.1 Monit

如果没有现成的监控系统，那么Monit（http://mmonit.com/monit/）将是一个免费的选择。Monit开发者提供了一个付费版，可以提供更多你或许会用到的功能。即便是免费版也提供了检测Flume代理是否运行的功能，如果没有运行可以重启，会发送电子邮件，这样你就能知道进程为何会死掉。Monit还提供了更多功能，不过这里将会主要介绍监控功能。如果你很聪明，那么除了本章介绍的功能外，你还可以添加对磁盘、CPU与内存使用情况的检测。

7.1.2 Nagios

还可以使用Nagios（http://www.nagios.org/）监控Flume代理进程。就像Monit一样，你可以配置Nagios来监控Flume代理并通过WebUI、电子邮件或是SNMP发出警告。不过，它没有提供重启功能。当然了，社区是足够强大的，现在已经有了很多针对其他应用的插件。我所在的公司就使用它来检测Hadoop Web UI的可用性。虽然没有提供应用健康状况的全貌，但它却对我们的Hadoop生态系统的整体监控提供了很多信息。

重申一次，如果公司已经有了类似的工具，那么在引入新工具前请先看看是否能够重用。

7.2　监控性能度量情况

既然已经介绍了用于进程监控的一些选择，那么你怎么知道应用是否如期望那样工作呢？在很多场合下，我看到貌似还在运行中的 syslog-ng 进程卡住了，不过它仅仅是没有发送任何数据而已。这里并非特指 syslog-ng，在出现一些没有处理的情况时，所有软件都将如此。

对于 Flume 数据流来说，你需要监控如下内容：

- 数据以期望的速度进入源中
- 数据没有超出通道的限制
- 数据以期望的速度离开接收器

Flume 提供了可插拔的监控框架，不过正如在本章一开始所述的那样，这项工作依旧在进行当中。这并不意味着你无法使用它。实际上，这表示在升级时你需要花点时间进行一些测试和集成。

虽然 Flume 文档并未提及，不过一般情况下都要在 Flume JVM 中开启 JMX（http://bit.ly/javajmx），使用 Nagios JMX 插件（http://bit.ly/nagiosjmx），当 Flume 代理的性能出现问题时发出警报。

7.2.1　Ganglia

对于监控 Flume 内部度量来说可以使用的一个监控选择就是 Ganglia 集成。Ganglia（http://ganglia.sourceforge.net/）是个开源的监控工具，用于收集度量、展示图形，并且可以分层以处理大型系

统。要想向 Ganglia 集群发送 Flume 度量，你需要在代理启动时向其传递几个属性：

Java 属性	值	说明
flume.monitoring.type	ganglia	将值设为 ganglia
flume.monitoring.hosts	host1:port1, host2:port2	为 gmond 进程指定的一个逗号分隔的主机：端口号对列表
flume.monitoring.pollInterval	60	两次发送数据之间间隔的秒数（默认是 60 秒）
flume.monitoring.isGanglia3	false	如果使用老式的 ganglia 3 协议则设为 true。默认是使用 v3.1 协议发送

查看同一个网络广播域（因为可达性是基于多路广播的）中的每一个 gmond 实例，找到 gmond.conf 中定义的 udp_recv_channel 块。我有两个相邻的服务器，具有如下两个相应的配置块：

```
udp_recv_channel {
  mcast_join = 239.2.14.22
  port = 8649
  bind = 239.2.14.22
  retry_bind = true
}
udp_recv_channel {
  mcast_join = 239.2.11.71
  port = 8649
  bind = 239.2.11.71
  retry_bind = true
}
```

在该示例中，第一台服务器的 IP 与端口号是 239.2.14.22/8649，第二台服务器则是 239.2.11.71/8649，启动属性如下：

```
-Dflume.monitoring.type=gangla
-Dflume.monitoring.hosts=239.2.14.22:8649,239.2.11.71:8649
```

这里使用默认的轮询间隔以及更新的 ganglia 协议。

虽然Ganglia支持通过TCP来接收数据，不过当前的Flume/Ganglia集成只支持通过多路广播UDP来发送数据。如果存在大型或是复杂的网络环境，那么当实际情况与期望不符时你需要咨询一下网络工程师。

7.2.2 内部HTTP服务器

可以配置Flume代理来开启输出JSON的HTTP服务器，外部则可以查询JSON数据。与Ganglia集成不同的是，有些外部实体需要调用Flume代理来轮询数据。从理论上来说，你可以使用Nagios轮询该JSON数据并在某些情况下发出警告，不过我自己从来没有使用过。当然了，这对于开发与测试来说是非常有用的，特别是在编写自定义Flume组件来确保它们生成的是有效度量的情况下更是如此。下表列出了可以在Flume代理启动时设置的Java属性：

Java属性	值	说明
flume.monitoring.type	http	设为http
flume.monitoring.port	端口号	绑定到HTTP服务器的端口号

用于度量的URL如下所示：

```
http://SERVER_OR_IP_OF_AGENT:PORT/metrics
```

这用于如下的Flume配置：

```
agent.sources = s1
agent.channels = c1
gent.sinks = k1
agent.sources.s1.type=avro
agent.sources.s1.bind=0.0.0.0
agent.sources.s1.port=12345
agent.sources.s1.channels=c1
agent.channels.c1.type=memory
```

```
agent.sinks.k1.type=avro
agent.sinks.k1.hostname=192.168.33.33
agent.sinks.k1.port=9999
agent.sinks.k1.channel=c1
```

此外，还有如下启动参数：

```
-Dflume.monitoring.type=http
-Dflume.monitoring.port=44444
```

访问 http://SERVER_OR_IP:44444/metrics，你会看到如下内容：

```
{
  "SOURCE.s1":{
    "OpenConnectionCount":"0",
    "AppendBatchAcceptedCount":"0",
    "AppendBatchReceivedCount":"0",
    "Type":"SOURCE",
    "EventAcceptedCount":"0",
    "AppendReceivedCount":"0",
    "StopTime":"0",
    "EventReceivedCount":"0",
    "StartTime":"1365128622891",
    "AppendAcceptedCount":"0"},
  "CHANNEL.c1":{
    "EventPutSuccessCount":"0",
    "ChannelFillPercentage":"0.0",
    "Type":"CHANNEL",
    "StopTime":"0",
    "EventPutAttemptCount":"0",
    "ChannelSize":"0",
    "StartTime":"1365128621890",
    "EventTakeSuccessCount":"0",
    "ChannelCapacity":"100",
    "EventTakeAttemptCount":"0"},
  "SINK.k1":{
    "BatchCompleteCount":"0",
    "ConnectionFailedCount":"4",
    "EventDrainAttemptCount":"0",
    "ConnectionCreatedCount":"0",
    "BatchEmptyCount":"0",
```

```
  "Type":"SINK",
  "ConnectionClosedCount":"0",
  "EventDrainSuccessCount":"0",
  "StopTime":"0",
  "StartTime":"1365128622325",
  "BatchUnderflowCount":"0"}
}
```

如你所见，该度量数据将每一个源、接收器与通道都分隔开来。每一种类型的源、通道与接收器都提供了一套自己的度量键，其中有一些共性，请寻找自己感兴趣的部分。比如，该 Avro 源有 OpenConnectionCount，它指的是连接的客户端数量（很大可能是发送数据的）。这有助于你确定依赖该数据的客户端数量是否与预期一致，或是客户端太多而需要开始对代理进行分层。

一般来说，可以通过通道的 ChannelSize 或 ChannelFillPercentage- 了解到数据的进入速度是不是大于出去速度；还可以了解到是否将其设置得足够大以满足数据维护 / 中断的需求。

来看看接收器，通过 EventDrainSuccessCount 与 EventDrainAttemptCount 可以了解到相对于尝试的次数来说，输出成功的频率有多大。在该示例中，我将一个 Avro 接收器配置到一个不存在的目标。如你所见，ConnectionFailedCount 值在不断增长，它能很好地展示出持续连接问题。甚至不断增加的 ConnectionCreatedCount 都会表明连接丢失以及重开启过于频繁。

实际上，除了观测 ChannelSize 与 ChannelFillPercentage 外并没有什么固定准则。每个用例都有自己的性能状况，因此要从小处着手，搭建监控，并在这一过程中不断学习。

7.2.3 自定义监控钩子

如果已经有了监控系统，那么你可能想花时间开发自定义的监控

报告机制。你可能以为这就像是实现 org.apache.flume.instrumentation.MonitorService 接口一样简单。确实需要这么做，不过看看接口，你会发现只有 start() 和 stop() 这两个方法。与更为直观的拦截器机制不同，代理期望你的 MonitorService 实现会启动 / 停止一个基于期望或配置的间隔时间来发送数据的线程（如果其是向接收服务发送数据的代理类型）。如果要操纵服务，比如 HTTP 服务，那么 start/stop 将用于开启或是停止监听服务。在内部，度量本身是通过各种源、接收器、通道与拦截器（使用以 org.apache.flume 开头的对象名）等发布到 JMX 的。你的实现需要从 MBeanServer 中读取这些信息。如果要实现自己的，那么我能给出的最佳建议是看看两个现有的实现源代码，然后自己再去实现。要想使用你的监控钩子，请将 flume.monitoring.type 属性设为实现类的全名。每当有新的 Flume 版本出现时，我们还得修改自己编写的钩子，直到框架逐步成熟并稳定下来为止。

7.3 小结

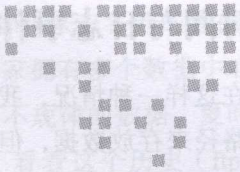
本章介绍了从进程级别以及内部度量级别来监控 Flume 代理的方式。

Monit 与 Nagios 是开源的进程监控选择。

接下来介绍了如何通过 Apache Flume 的 HTTP 实现，凭借 Ganglia 与 JSON 完成 Flume 代理的内部监控度量。

最后介绍了如果想要集成 Flume 默认情况下不支持的其他工具时该如何集成自定义的监控实现。

下一章将会介绍 Flume 部署的一般原则。



第 8 章 Chapter 8

万法皆空——实时分布式数据收集的现状

在本章中我打算围绕着 Hadoop 的数据收集主题谈一些不那么具体、随性的想法。这背后并没有什么自然科学作为支撑，因此你完全可以不同意我的看法。

虽然 Hadoop 是消费海量数据的绝佳工具，不过我脑海中经常浮现出 1886 年明尼苏达州圣克罗伊河堵塞的场景（http://www.nps.gov/sacn/historyculture/stories.htm）。在处理海量数据时，你要确保自己的河流不会出现堵塞的情况。请严肃对待上一章介绍的关于监控的主题，这绝不是什么玩笑。

8.1 传输时间与日志事件

有时会存在这样一种情况，我们根据文件名中的日期模式或是HDFS中的路径来存放数据，但它与目录的内容并不匹配。我们希望2013/03/29中的数据包含这一天的所有数据。不过实际情况却是该日期是从传输中获取的。我们使用的syslog版本会对头进行重写，包括日期部分，这会导致传输期间获取到的数据没有反映出记录的原始时间。通常情况下，这个影响不大，只不过是一两秒而已，因此没人会注意到它。如果有一天某台中继服务器宕机了，当在上游服务器阻塞的数据最后发出时，它持有的是当前时间。在这种情况下，时间偏移量可能会有好几天，这真是一团糟。

如果根据日期来存放数据，那么请确保这种情况不会发生在你身上。请检查日期的边际情况，看看是否如你预期那样，请对中断场景进行测试以防止在生产环境下发生这些问题。

如前所述，由于计划中或是计划外的维护（甚至可能是因为短暂的网络间断），这些中继很有可能会导致发送重复或是过期的事件，请确保在处理原始数据时考虑到这一点。Flume并不提供发送/顺序的保证。如果需要，请使用事务型数据库。

8.2 万恶的时区

如果没有注意到第4章中关于本地时间的论述，那么在这里我再重复一遍——时区是万恶的，就像Dr.Evil（http://en.wikipedia.org/wiki/Dr._Evil）那样万恶；大家也别忘记其“Mini Me”（http://en.wikipedia.org/wiki/Mini-Me）对应者——夏时制。

我们生活在全球化时代，你将来自各处的数据拉到 Hadoop 集群中。你甚至可能在不同的国家拥有多个数据中心。在分析数据时你最不想做的事情就是处理不规律的数据。夏时制在一年中会改变地球上多个地区的时间，看看这个历史（ftp://ftp.iana.org/tz/releases/）。别自寻烦恼了，使用 UTC 吧。你也可以将其转换为可读性良好的“本地时间”。不过如果是在集群上，那么请一定要使用 UTC。可以通过下面这个 Java 启动参数使用 UTC（如果无法在系统范围内设定）：

```
-Duser.timezone=UTC
```

我住在芝加哥，使用的计算机用的是中央标准时间，它针对夏时制进行了调整。在我们的 Hadoop 集群中，我们将数据设定为 YYYY/MM/DD/HH 布局。一年两次，有些东西会打破规则。在秋季，2 a.m 目录中的数据将是平时的两倍。在春季则没有 2 a.m 目录，这简直太疯狂了！

8.3　容量规划

无论你认为自己的数据有多少，情况总是会随着时间的流逝而发生变化。新项目会出现，现有项目的数据创建率也会发生变化（上升或是下降）。数据量经常会在一天内起起落落。最后，构成 Hadoop 集群的服务器数量也会不断发生变化。

对于 Hadoop 集群要有多少额外的存储容量这个问题存在着几种观点（我们使用的是 20%，这个值没有什么科学依据——这意味着当有 80% 的容量已满时我们才会采购硬件，不过这并不会造成什么影响，一直到 85% 到 90% 的利用率时才应该引起我们的高度注意）。

你可能还需要在单个代理中创建多个流。目前，源与接收处理器是单线程的，因此在大数据量的情况下，调节批量大小会受到限制。

我们应该根据实际数量来调节供给 Hadoop 的 Flume 代理数量。观察通道数量，看看在正常负载下写入操作是否保持正常。可以通过调节最大的通道容量来处理各种负载情况。办法总是很多的，不过一次长期停机会导致预算超出最保守的估计。这时就要选择最重要的数据并调节通道容量来反映这种情况。也就是说，如果超出限制，不那么重要的数据将会被丢弃。

公司的钱不是无穷无尽的，在某一点上，数据的价值与持续扩展集群的成本将不成比例。这也是我为何说为收集的数据量设定限制值是非常重要的原因所在。向 Hadoop 发送数据的任何项目都会说它的数据有多么重要，如果删除了旧数据将会产生多大的损失。开支票的那些人也只能通过这种方式了解情况。

8.4 多数据中心的注意事项

如果业务用到了多数据中心，并且收集到了海量的数据，那么你可能希望在每个数据中心都搭建一个 Hadoop 集群而不是将所有收集到的数据都发送给单个数据中心。这样做会导致数据分析变得困难，因为无法对所有数据只运行一个 MapReduce job。与之相反，你需要运行并行的 job，然后将结果合并起来。可以对搜索与统计问题采用这种方式，但对于取平均数则不行，因为平均数的平均数与平均数是不同的。

将所有数据都拉取到单个集群中还可能会超出网络的处理能

力。根据数据中心彼此连接方式的不同，你可能无法传输所需的数据量。最后，如果集群全部失败或是损坏则会将一切都清空，因为大多数集群常常由于太大而无法对全部数据进行备份（除了一些高价值的数据外）。在这种情况下，保留一些老数据要比什么都没有强。对于多个 Hadoop 集群来说，你可以通过故障恢复接收处理器将数据转发到不同集群上（如果不想等待再将其发送到本地）。

如果选择将所有数据都发送到单个目的地，那么请考虑添加一个具有大容量磁盘的机器作为数据中心的中继服务器。通过这种方式，如果出现了通信问题或是扩展集群维护，那么你可以将数据堆积到一台机器上，这台机器与为客户提供服务的机器不同。即便在单个数据中心中这种做法也是有益的。

8.5　合规性与数据失效

记住，公司收集的关于客户的数据可能会包含敏感信息。在访问这些数据时你可能会受到其他一些监管限制，如 Payment Card Industry（PCI—http://en.wikipedia.org/wiki/PCI_DSS）或萨班斯法案（SOX—http://en.wikipedia.org/wiki/Sarbanes%E2%80%93Oxley_Act）。如果没有对集群中这些数据的访问进行恰当的处理，那么政府就会找上门来，更糟的是，如果客户认为你没有保护他们的权利和身份，那么他们就不会再找你了。请对个人信息进行加工和处理。你所寻找的商业信息更多的是诸如“查找锤子的用户有多少人真正购买了？”而不是“有多少客户叫 Bob？”。正如第 6 章所述，我们可以很轻松地编写一个拦截器来加工处理个人身份信息（PII—http://en.wikipedia.org/wiki/Personally_identifiable_information），这样

在移动这些数据时就不会出现合规性问题了。

你所在的公司可能有文档化的保存策略，很有可能包括放到Hadoop中的数据。请确保将政策中明确指明的不能保留的数据删除掉，否则等待你的将是律师的召见。

8.6 小结

本章介绍了在规划Flume实现时需要考虑的几个现实问题，主要包括：

- 传输时间并不总是与事件时间相匹配
- 夏时制对基于时间的逻辑所引发的问题
- 容量规划考量
- 拥有多个数据中心时要考虑的问题
- 数据合规性
- 数据失效性

下篇 Part 2

MapReduce 模式

- 第 9 章　使用 Java 编写一个单词统计应用（初级）
- 第 10 章　使用 MapReduce 编写一个单词统计应用并运行（初级）
- 第 11 章　在分布式环境中安装 Hadoop 并运行单词统计应用（初级）
- 第 12 章　编写格式化器（中级）
- 第 13 章　分析——使用 MapReduce 绘制频度分布（中级）
- 第 14 章　关系操作——使用 MapReduce 连接两个数据集（高级）
- 第 15 章　使用 MapReduce 实现集合操作（中级）
- 第 16 章　使用 MapReduce 实现交叉相关（中级）
- 第 17 章　使用 MapReduce 实现简单搜索（中级）
- 第 18 章　使用 MapReduce 实现简单的图操作（高级）
- 第 19 章　使用 MapReduce 实现 Kmeans（高级）

下篇介绍了Hadoop及几种基于Hadoop的分析实现，旨在为初学者提供一份简明的Hadoop实践指南。

从历史上来看，数据处理完全是通过数据库技术来实现的。大多数数据都有定义良好的结构，通常会存储在数据库中。在处理这类数据时，关系数据库是最常见的存储选择。这些数据集很小，可以通过关系数据库存储和查询。

然而，数据集开始逐渐变大。很快，Google等高科技公司发现很多大数据集并不适合于数据库。比如，Google一直在抓取并索引整个互联网，数据量很快就达到了TB和PB级别。Google开发了一个名为MapReduce的新的编程模型来处理大规模的数据分析，不久之后他们在论文“MapReduce: Simplified Data Processing on Large Clusters”中介绍了该模型。

Hadoop是个基于Java的开源项目，它是MapReduce编程模型的一个实现。在Hadoop的帮助下，用户只需编写处理逻辑，Hadoop等MapReduce框架可以在执行逻辑的同时处理分布式任务的方方面面，比如说作业调度、数据迁移与失败处理等，这一切对用户来说都是透明的。

Hadoop已经成为Java领域中事实上的MapReduce实现。从学生一直到大型企业等诸多用户都在使用Hadoop解决数据处理难题，MapReduce也成为就业市场上最为注重的技能之一。

本篇旨在对MapReduce进行一个简明的介绍，同时还会介绍可以通过MapReduce解决的各种难题。现在有很多资源介绍了如何上手Hadoop并运行字数统计示例，这相当于MapReduce世界中的“Hello World”。然而，在如何通过MapReduce解决各类难题

上却没有多少参考资料，本篇内容将试图填充这个鸿沟。

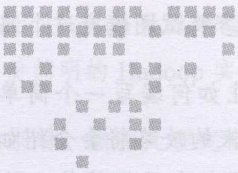
本篇的前3个攻略重点关注如何编写一个简单的MapReduce程序并使用Hadoop运行。接下来的攻略将会介绍如何编写自定义的格式化器来解析输入文件中的复杂数据结构。然后将会介绍如何通过MapReduce进行基本的分析、如何使用GUN plot来绘制结果。这是Hadoop常见的使用场景之一。

其余的攻略涵盖了MapReduce所能解决的各类问题，并给出了适合于该类问题的解决方案模式。所涵盖的问题种类有：集合操作、交叉相关、搜索、图与关系操作及相似性聚类。

我们在整篇中都会使用斯坦福大学所收集的关于Amazon销售数据的公开数据集。该数据集提供了关于图书以及购买图书的用户信息。下面展示了一个示例数据记录：

```
Id:   3
ASIN: 0486287785
title: World War II Allied Fighter Planes Trading Cards
group: Book
salesrank: 1270652
similar: 0
categories: 1
   |Books[283155]|Subjects[1000]|Home & Garden[48]|Crafts &
Hobbies[5126]|General[5144]
reviews: total: 1  downloaded: 1  avg rating: 5
    2003-7-10  cutomer: A3IDGASRQAW8B2  rating: 5  votes:   2
helpful:   2
```

数据集位于http://snap.stanford.edu/data/#amazon，大小约为1G。除非你能使用大规模的Hadoop集群，否则我建议你在运行示例时使用示例目录中相同数据集更小的子集。



Chapter 9 第 9 章

使用 Java 编写一个单词统计应用（初级）

该攻略介绍了如何使用基本的 Java 程序编写一个 Hadoop 的分析任务。接下来，我们会探讨在多台机器上运行程序所面临的挑战，从而引入 MapReduce 之类的框架。

该示例将会演示如何统计一个文件中单词出现的次数。

9.1 准备工作

该攻略假设你有一台装有 Java 的计算机，并且配置好了 JAVA_HOME 环境变量，指向了 Java 的安装位置。请下载本书代码并将其解压缩到某个目录。我们称这个解压缩后的目录为 SAMPLE_DIR。

9.2　操作步骤

1. 将 hadoop-microbook.jar 中的数据集复制到 HADOOP_HOME。

2. 运行字数统计程序，方式是从 HADOOP_HOME 中执行如下命令：

```
$ java -cp hadoop-microbook.jar microbook.wordcount.JavaWordCount
SAMPLE_DIR/amazon-meta.txt results.txt
```

3. 程序将会运行，并将输入文件的单词数写到名为 results.txt 的文件中。其输出结果如下所示：

```
B00007ELF7=1
Vincent[412370]=2
35681=1
```

9.3　示例说明

该攻略的源代码位于 src/microbook/JavaWordCount.java。代码会逐行读取文件，对每一行进行分词，然后计算每个单词出现的次数。

```
BufferedReaderbr = new BufferedReader(
newFileReader(args[0]));
String line = br.readLine();
while (line != null) {
    StringTokenizertokenizer = new StringTokenizer(line);
    while(tokenizer.hasMoreTokens()){
        String token = tokenizer.nextToken();
        if(tokenMap.containsKey(token)){
            Integer value = (Integer)tokenMap.get(token);
            tokenMap.put(token, value+1);
        }else{
            tokenMap.put(token, new Integer(1));
        }
    }
    line = br.readLine();
```

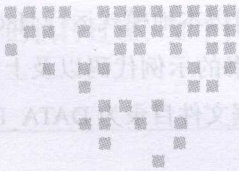
```
}

Writer writer = new BufferedWriter(
    new FileWriter("results.txt"));

for(Entry<String, Integer> entry: tokenMap.entrySet()){
    writer.write(entry.getKey() + "= "+ entry.getValue());
}
```

该程序只能使用一台计算机进行处理。对于大小适量的数据集来说，这是可以接受的。然而，对于大型数据集来说，这么做要花费大量时间。此外，该解决方案将所有数据都放在内存中，对于大型数据集来说，程序很有可能会出现内存溢出的问题。为了避免这个问题，当可用的内存有限时，程序需要将一些数据移动到磁盘上，而这么做又会进一步减缓程序的速度。

我们通过多台计算机来解决大型数据集的问题，使用这些计算机并行处理数据集。然而，编写一个能在分布式环境下处理数据集的程序可不是那么轻松的事情。这种程序将会面临如下挑战：

- 分布式程序需要找到可用的机器，并为这些机器分配工作。
- 程序需要通过消息传递或是共享文件系统等方式在机器间传输数据。这种框架要能做到可集成、可配置且可维护。
- 程序要能检测到失败并采取校正动作。
- 程序要能确保为所有节点都分配了大致相同的工作量，这样才能保证资源的利用率达到最佳。
- 程序要能检测到执行的结束，然后收集所有结果并将其传递给最终位置。

虽然可以编写这样的程序，不过一遍又一遍地写这样的程序是很浪费时间的。凭借 Hadoop 等基于 MapReduce 的框架，用户只需要编写处理逻辑即可，框架会负责分布式执行的所有复杂工作。



第 10 章 Chapter 10

使用 MapReduce 编写一个单词统计应用并运行（初级）

第一个攻略介绍了如何在不使用 MapReduce 的情况下实现字数统计应用，同时也谈到了这种实现方式的局限性。本攻略将会介绍如何通过 MapReduce 实现一个字数统计应用并解释其原理。

10.1 准备工作

1. 本攻略假定你有一台安装了 Java 开发包（JDK）的计算机，并且配置好了 JAVA_HOME 变量。

2. 从 http://hadoop.apache.org/releases.html 页面下载 Hadoop 分发包 1.1.x。

3. 解压缩下载的分发包；我们称解压缩后的目录为 HADOOP_

HOME。现在可以本地模式运行 Hadoop job 了。

4. 下载本书的示例代码以及上一个攻略中所介绍的数据文件，我们称这个数据文件目录为 DATA_DIR。

10.2 操作步骤

1. 将 hadoop-microbook.jar 文件从 SAMPLE_DIR 复制到 HADOOP_HOME。

2. 在 HADOOP_HOME 下通过如下命令运行 MapReduce job：

```
$bin/hadoop -cp hadoop-microbook.jar  microbook.wordcount.
WordCount amazon-meta.txt wordcount-output1
```

3. 可以在输出目录中查看到结果。

4. 结果如下所示：

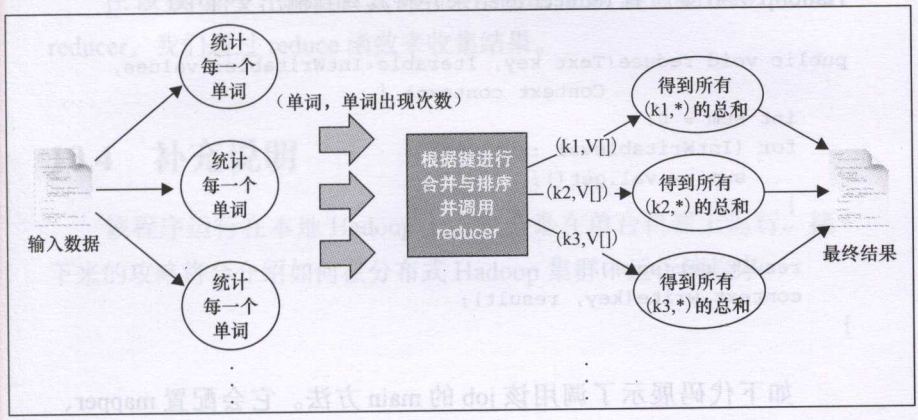
```
B00007ELF7=1
Vincent[412370]=2
35681=1
```

10.3 示例说明

本攻略的源代码位于 src/microbook/wordcount/WordCount.java。

该字数统计 job 接收一个输入目录、一个 mapper 函数和一个 reducer 函数作为输入。我们使用 mapper 函数并行处理数据，使用 reducer 函数收集 mapper 的结果并生成最终的结果。mapper 会通过一个基于键值的模型将其结果发送给 reducer。下面来详细介绍 MapReduce 的执行过程。

下图描述了 MapReduce job 的执行过程，如下代码清单展示了 mapper 与 reducer 函数：



在运行 MapReduce job 时，Hadoop 首先会从输入目录中逐行读取输入文件。接下来，Hadoop 会对每一行调用一次 mapper 并将该行作为参数传递进去。随后，每个 mapper 会解析该行，并将接收到的每一行中的单词提取出来作为输入。处理完毕后，mapper 会将单词及单词数发送给 reducer，这是通过将单词与单词数作为名值对发送出去而实现的。

```
public void map(Object key, Text value, Context context) {
    StringTokenizeritr = new
        StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
       word.set(itr.nextToken());
       context.write(word, one);
    }
}
```

Hadoop 会收集 mapper 函数发送出去的所有名值对，然后根据键进行排序。这里的键指的是单词，值指的是单词出现的次数。接

下来会针对每个键调用一次reducer，并将相同键的所有值作为参数传递进去。reducer会计算这些值的总和并根据键再次将其发送出去。Hadoop会收集所有reducer的结果并将其写到输出文件中。

```
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context) {
    int sum = 0;
    for (IntWritableval : values) {
        sum += val.get();
    }

    result.set(sum);
    context.write(key, result);
}
```

如下代码展示了调用该job的main方法。它会配置mapper、reducer、输入与输出格式，以及输入与输出目录。mapper与reducer的输入和输出分别对应于setOutputKeyClass(..)、setOutputValueClass(..)、job.setMapOutputKeyClass(..)与job.setMapOutputValueClass(..)设定的值：

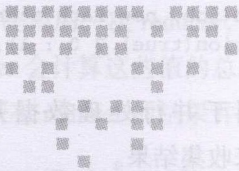
```
JobConfconf = new JobConf();
String[] otherArgs =
    new GenericOptionsParser(conf, args).getRemainingArgs();
if (otherArgs.length != 2) {
    System.err.println("Usage: <in><out>");
    System.exit(2);
}
Job job = new Job(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(WordcountMapper.class);
job.setReducerClass(WordcountReducer.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
```

```
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

在该模型中，map 函数用于并行处理数据并将其分发给 reducer，我们通过 reduce 函数来收集结果。

10.4　补充说明

该程序运行在本地 Hadoop 上，完全是在单台机器上运行。接下来的攻略将会介绍如何在分布式 Hadoop 集群中运行该程序。



Chapter 11 第 11 章

在分布式环境中安装 Hadoop 并运行单词统计应用（初级）

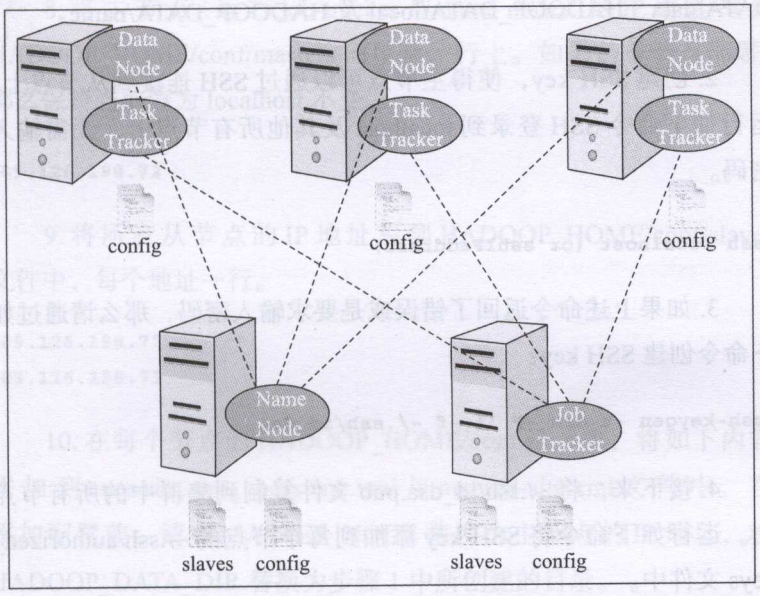
下图展示了一个典型的 Hadoop 部署图。一个 Hadoop 部署包含了一个 name node、多个 data node、一个 job tracker，以及多个 task tracker。下面介绍每一种节点。

Name node 与 data node 向 HDFS 文件系统提供了这样一些信息，其中 data node 持有实际的数据，name node 持有文件位于哪个 data node 上的信息。如果用户想读取文件，那么首先需要与 name node 交互，找到文件的位置，然后通过 data node 访问文件。

与之类似，job tracker 会追踪 MapReduce job 并调度 Task Tracker 中的 map 与 reduce 任务。用户将 job 提交到 Job Tracker，后者在 Task Tracker 中运行这些 job。然而，我们也可以在单个或是多个节

点中运行所有类型的服务器。

本攻略将会介绍如何创建自己的 Hadoop 集群。我们需要配置 job tracker 与 task tracker，然后在 job tracker 的 HADOOP_HOME/conf/slaves 文件中指向 task tracker。在启动 job tracker 时，它会启动 task tracker 节点。下面来看看详细的部署：



11.1　准备工作

1. 至少需要一台 Linux 或是 Mac OS X 机器，可以使用单台或是多台机器。如果使用多台机器，你应该选择一台机器作为主节点，将其他节点作为从节点。需要在主节点上运行 HDFS name node 与 job tracker。如果使用单台机器，那么它既作为主节点也作为从节点。

2. 在需要配置 Hadoop 的所有机器上安装 Java。

11.2 操作步骤

1. 在每台机器上为 Hadoop 数据创建一个目录，我们称这个目录为 HADOOP_DATA_DIR。接下来创建 3 个子目录：HADOOP_DATA/data、HADOOP_DATA/local 及 HADOOP_DATA/name。

2. 创建 SSH key，使得主节点可以通过 SSH 连接到从节点上。运行如下命令 SSH 登录到 localhost 及其他所有节点上，无需输入密码。

```
>ssh localhost (or sshIPaddress)
```

3. 如果上述命令返回了错误或是要求输入密码，那么请通过如下命令创建 SSH key：

```
>ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
```

4. 接下来，将 ~/.ssh/id_dsa.pub 文件复制到集群中的所有节点上。运行如下命令将 SSH key 添加到每个节点的 ~/.ssh/authorized_keys 文件中。

```
>cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

5. 通过如下命令登录：

```
>ssh localhost
```

6. 通过如下命令将 Hadoop 分发包解压缩到所有机器的相同位置处：

```
>tar -zxvf hadoop-1.0.0.tar.gz.
```

7. 在所有机器上编辑 HADOOP_HOME/conf/hadoop-env.sh 文件，取消 JAVA_HOME 一行的注释，并将其指向本地的 Java 目录。比如，如果 Java 位于 /opt/jdk1.6，那么将这一行修改为 export JAVA_HOME=/opt/jdk1.6。

8. 将主节点（运行 job tracker 与 name node）的 IP 地址写到 HADOOP_HOME/conf/masters 的单独一行上。如果是单节点部署，那么保持当前值为 localhost 不变。

```
209.126.198.72
```

9. 将所有从节点的 IP 地址写到 HADOOP_HOME/conf/slaves 文件中，每个地址一行。

```
209.126.198.72
209.126.198.71
```

10. 在每个节点的 HADOOP_HOME/conf 目录中，将如下内容添加到 core-site.xml、hdfs-site.xml 与 mapred-site.xml 文件中。在添加配置前，请将 MASTER_NODE 替换为主节点的 IP 地址，将 HADOOP_DATA_DIR 替换为步骤 1 中所创建的目录。

11. 将 name node 的 URL 添加到 HADOOP_HOME/conf/core-site.xml 中，如下代码所示：

```
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://MASTER_NODE:9000/</value>
</property>
</configuration>
```

12. 在 HADOOP_HOME/conf/hdfs-site.xml 中添加存储元数据（names）与数据的位置，如下代码所示：

```
<configuration>
<property>
<name>dfs.name.dir</name>
<value>HADOOP_DATA_DIR/name</value>
</property>
<property>
<name>dfs.data.dir</name>
<value>HADOOP_DATA_DIR/data</value>
</property>
</configuration>
```

13. MapReduce 本地目录是 Hadoop 用来存储临时文件的目录。将 job tracker 位置添加到 HADOOP_HOME/conf/mapred-site.xml 中。Hadoop 客户端在提交 job 时会使用该 job tracker。最后的属性设置每个节点最大的 map 任务数，应该将其设为机器上的 CPU 核心数。

```
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>MASTER_NODE:9001</value>
</property>
<property>
<name>mapred.local.dir</name>
<value>HADOOP_DATA_DIR/local</value>
</property>
<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value>8</value>
</property>
</configuration>
```

14. 从 Hadoop name node（又叫做主节点）执行如下命令来格式化一个新的 HDFS 文件系统。

```
>run bin/hadoopnamenode -format
...
/Users/srinath/playground/hadoop-book/hadoop-temp/dfs/name has
been successfully formatted.
```

15. 在主节点上，将目录修改为 HADOOP_HOME 并执行如下命令：

```
>bin/start-dfs.sh
>bin/start-mapred.sh
```

16. 通过 ps|grep java 命令列出进程来验证安装。主节点会列出 3 个进程：name node、data node 与 job tracker，task tracker 与从节点有一个 data node 和 task tracker。

17. 浏览基于 Web 的监控页面来查看 name node 与 job tracker，NameNode——http://MASTER_NODE:50070/，JobTracker——http://MASTER_NODE:50030/。

18. 日志文件位于 ${HADOOP_HOME}/logs 中。

19. 通过 HDFS 命令行列出文件以确保 HDFS 安装正确。

```
bin/hadoopdfs -ls /
Found 2 items
drwxr-xr-x   - srinathsupergroup     0 2012-04-09 08:47 /Users
drwxr-xr-x   - srinathsupergroup     0 2012-04-09 08:47 /tmp
```

20. 从 http://snap.stanford.edu/data/bigdata/amazon/amazon-meta.txt.gz 下载网络日志数据集并解压缩，我们称其为 DATA_DIR。该数据集大约有 1G 大小，如果希望执行速度能快一些，那么请使用该数据集的一个子集。

21. 将 hadoop-microbook.jar 文件从 SAMPLE_DIR 复制到 HADOOP_

HOME。

22. 如果没有做的话，那么请通过如下命令将亚马逊数据集上传到 HDFS 文件系统：

```
>bin/hadoopdfs -mkdir /data/
>bin/hadoopdfs -mkdir /data/amazon-dataset
>bin/hadoopdfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-dataset/
>bin/hadoopdfs -ls /data/amazon-dataset
```

23. 从 HADOOP_HOME 执行如下命令来运行 MapReduce job：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.wrodcount.WordCount /data/amazon-dataset /data/wordcount-doutput
```

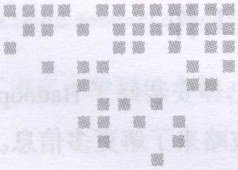
24. 可以在输出目录中找到 MapReduce job 的结果。通过如下命令列出内容：

```
$ bin/hadoop jar hadoop-microbook.jar dfs -ls /data/wordcount-doutput
```

11.3 示例说明

在之前的介绍中曾提到过，Hadoop 安装包含了 HDFS 节点、一个 job tracker 与 worker node。在启动 name node 时，它会通过 HADOOP_HOME/salves 文件找到从节点，并使用 SSH 启动远程服务器上的 data node。在启动 job tracker 时，它会通过 HADOOP_HOME/salves 文件找到从节点并启动 task tracker。

当运行 MapReduce job 时，客户端会通过配置找到 job tracker，并将 job 提交到 job tracker。客户端会等待执行完成并接收标准输出，在 job 运行时会将其打印到控制台上。



第 12 章 Chapter 12

编写格式化器（中级）

默认情况下，在运行 MapReduce job 时，它会逐行读取输入文件并将每一行内容传递给 map 函数。对于大多数情况来说，这么做没问题。然而，有时一条数据记录会在多行当中。比如，之前曾提到过，我们的数据集有一个记录格式会跨越多行。在这种情况下，编写一个 MapReduce job 将这些行放到一起并对其进行处理是非常复杂的。

好消息是我们可以覆写 Hadoop 读写文件的方式，这样你就可以控制处理步骤了。我们可以通过添加新的格式化器来做到这一点。本攻略将会介绍如何编写新的格式化器。

格式化器的代码位于 src/microbook/ItemSalesDataFormat.java 中。本攻略将会使用该格式化器从数据集中读取记录，然后计算书名中的单词数。

12.1 准备工作

1. 假定你已经安装好了 Hadoop 并启动。请参考第 9 章和第 11 章介绍的两个攻略来了解更多信息。我们使用 HADOOP_HOME 指代 Hadoop 安装目录。

2. 本攻略假定你了解 Hadoop 的处理方式。如果不了解，那么阅读第 10 章。

3. 下载本章的示例代码，复制攻略使用第 10 章中提到的数据文件。

12.2 操作步骤

1. 如果之前没有做过，那么请通过如下命令将亚马逊数据集上传到 HDFS 文件系统。

```
>bin/hadoopdfs -mkdir /data/
>bin/hadoopdfs -mkdir /data/amazon-dataset
>bin/hadoopdfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-dataset/
>bin/hadoopdfs -ls /data/amazon-dataset
```

2. 将 hadoop-microbook.jar 文件从 SAMPLE_DIR 复制到 HADOOP_HOME。

3. 从 HADOOP_HOME 通过如下命令运行 MapReduce job：

```
>bin/hadoop jar hadoop-microbook.jar  microbook.format.
TitleWordCount /data/amazon-dataset /data/titlewordcount-output
```

4. 可以通过如下命令在输出目录中查看结果：

```
>bin/Hadoop dfs -cat /data/titlewordcount-output/*
```

你会看到结果中已经记录了书名中单词的个数：

12.3　示例说明

在该攻略中，我们执行了一个 MapReduce job，使用自定义格式化器来解析数据集。通过将如下加粗的行添加到主程序中来启用格式化器。

```
JobConfconf = new JobConf();
String[] otherArgs =
    new GenericOptionsParser(conf, args).getRemainingArgs();
if (otherArgs.length != 2) {
    System.err.println("Usage: wordcount<in><out>");
    System.exit(2);
}
Job job = new Job(conf, "word count");
job.setJarByClass(TitleWordCount.class);
job.setMapperClass(WordcountMapper.class);
job.setReducerClass(WordcountReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(ItemSalesDataFormat.class);

FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

如下代码清单实现了格式化器：

```
public class ItemSalesDataFormat
    extends FileInputFormat<Text, Text>{
    private ItemSalesDataReadersaleDataReader = null;

    public RecordReader<Text, Text>createRecordReader(
        InputSplitinputSplit, TaskAttemptContext attempt) {
```

```
        saleDataReader = new ItemSalesDataReader();
        saleDataReader.initialize(inputSplit, attempt);
        return saleDataReader;
    }
}
```

该格式化器创建了一个记录读取器，它会完成大多数实际工作。当运行 Hadoop job 时，它会找到该格式化器，创建新的记录读取器并将每个文件传递进去，从记录读取器中读取记录，然后将这些记录传递给 map 任务。

如下代码清单展示了记录读取器：

```
public class ItemSalesDataReader
extendsRecordReader<Text, Text> {

  public void initialize(InputSplitinputSplit,
  TaskAttemptContext attempt) {
     //open the file
  }

  public boolean nextKeyValue(){
      //parse the file until end of first record
     }

  public Text getCurrentKey(){ ... }

  public Text getCurrentValue(){ ... }

  public float getProgress(){ ..    }

  public void close() throws IOException {
  //close the file
  }
}
```

Hadoop 会调用 initialize（..）方法并传递输入文件，然后再调用其他方法来读取键。当 nextKeyValue() 被调用时，实现会读取下

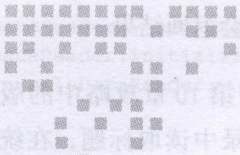
一条记录，当其他方法被调用时会返回结果。

这里的 mapper 和 reducer 与第 10 章攻略中的版本非常像，只不过 mapper 会从它接收到的记录中读取标题，在统计单词时只会用到标题。mapper 与 reducer 的代码位于 src/microbook/wordcount/TitleWordCount.java。

12.4　补充说明

Hadoop 还支持输出格式化器，你可以通过类似的方式开启它，它会返回一个 RecordWriter 而非读取器。可以通过文章 http://www.infoq.com/articles/HadoopOutputFormat 了解更多信息，也可以通过 Srinath Perera 与 Thilina Gunarathne 编写，Packt 免费发布的《Hadoop MapReduce Cookbook》一书中的部分章节进行学习，地址是 http://www.packtpub.com/article/advanced-hadoop-mapreduce-administration。

Hadoop 还提供了其他几个输入输出格式化器，比如 ComposableInputFormat、CompositeInputFormat、DBInputFormat、DBOutputFormat、IndexUpdateOutputFormat、MapFileOutputFormat、MultipleOutputFormat、MultipleSequenceFileOutputFormat、MultipleTextOutputFormat、NullOutputFormat、SequenceFileAsBinaryOutputFormat、SequenceFileOutputFormat、TeraOutputFormat 及 TextOutputFormat。在大多数情况下，你可以使用上述格式化器而没必要编写新的。



Chapter 13 第 13 章

分析——使用 MapReduce 绘制频度分布（中级）

很多时候，我们使用 Hadoop 进行分析计算，这是关于数据的一些基本统计。在这些情况下，我们通过 Hadoop 遍历数据，然后对数据进行一些分析。常见的分析如下所示：

- 计算数据集的一些统计属性，如最小值、最大值、平均值、中位数、标准差等。对于一个数据集来说，通常情况下会有多个维度（比如，在处理 HTTP 访问日志时，网页的名字、网页的大小、访问时间等，它们都是不同的维度）。我们可以通过一个或多个维度来度量出上述属性。比如，可以将数据划分到多个组中，然后计算出每种情况下的平均值。
- 频度分布直方图会统计数据集中每个条目的出现次数，对这些

频度进行排序，然后将不同的条目作为 X 轴，将频度作为 Y 轴。

- 找到两个维度间的关联关系（比如，访问数量与网络访问文件大小之间的关系）。
- 假设检验：通过给定的数据集来验证或是反证某个假设。

不过，Hadoop 只会生成数字。虽然数字包含了各种信息，但我们却不善于仅仅通过数字来洞察出总体的趋势。另一方面，人眼非常善于探测模式，对数据进行绘图会让我们对数据有更加深刻的理解。因此，我们常常会通过一些绘图程序对 Hadoop job 的结果进行绘制。

本攻略将会介绍如何通过 MapReduce 计算出每个客户购买的商品数量的频度分布。接下来会使用 gnuplot（一个免费且强大的绘图程序）对 Hadoop job 的结果进行绘制。

13.1　准备工作

1. 本攻略假定你的计算机上已经安装好 Java 并且配置好了 JAVA_HOME 变量。

2. 从 http://hadoop.apache.org/releases.html 页面下载 Hadoop 分发包 1.1.x。

3. 解压缩下载的分发包，我们称这个解压缩后的目录为 HADOOP_HOME。

4. 下载本章的示例代码，复制第 9 章中所提及的数据文件。

13.2　操作步骤

1. 如果之前没有做过，那么请通过如下命令将亚马逊数据集上

传到HDFS文件系统中。

```
>bin/hadoopdfs -mkdir /data/
>bin/hadoopdfs -mkdir /data/amazon-dataset
>bin/hadoopdfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-dataset/
>bin/hadoopdfs -ls /data/amazon-dataset
```

2. 将hadoop-microbook.jar文件从SAMPLE_DIR复制到HADOOP_HOME中。

3. 运行第1个MapReduce job来计算购买频度。为了做到这一点，请从HADOOP_HOME中运行如下命令：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.frequency.BuyingFrequencyAnalyzer/data/amazon-dataset /data/frequency-output1
```

4. 执行如下命令运行第2个MapReduce job对第1个MapReduce job的结果进行排序：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.frequency.SimpleResultSorter /data/frequency-output1 frequency-output2
```

5. 可以在输出目录中找到结果。通过如下命令将结果复制到HADOOP_HOME中：

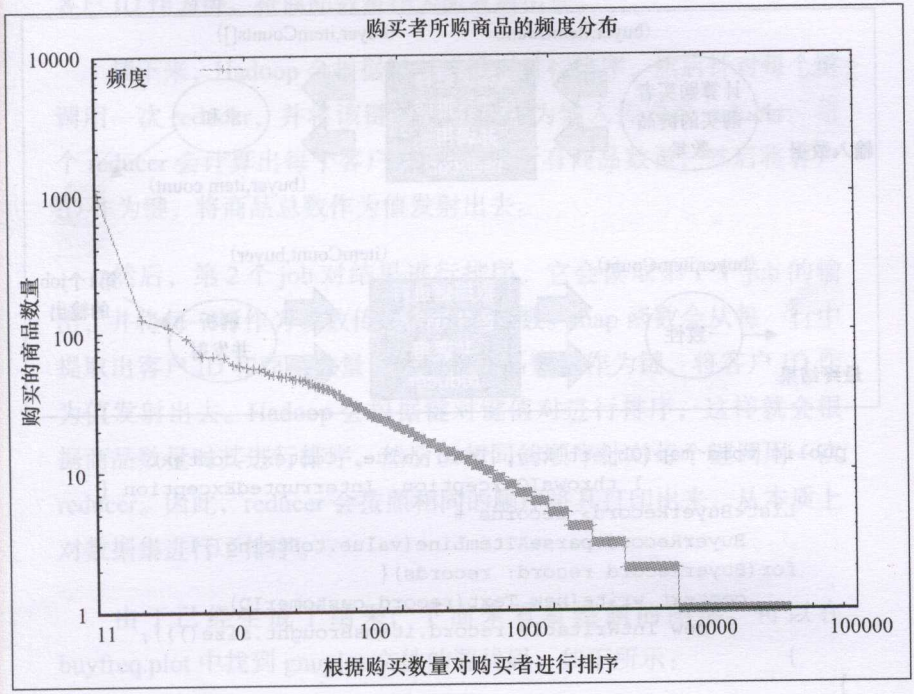
```
$ bin/Hadoop dfs -get /data/frequency-output2/part-r-00000 1.data
```

6. 将SAMPLE_DIR中所有的*.plot文件复制到HADOOP_HOME中。

7. 从HADOOP_HOME中运行如下命令来生成图例。

```
$gnuplot buyfreq.plot
```

8. 这会生成一个名为 buyfreq.png 的文件，如下图所示：



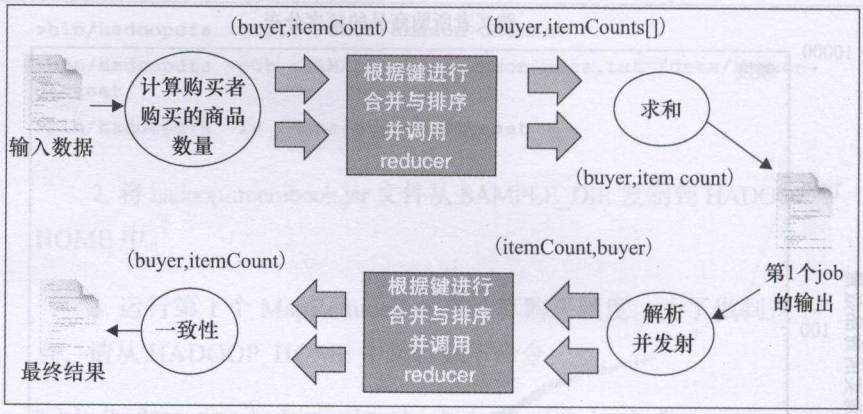
如图所示，有少量购买者购买了大量的商品。这个分布要比正态分布更加陡峭，我们称之为幂次分布。通过该示例可以看到分析与图形结果能够让我们更加深入地理解数据集中的底层模式。

13.3　示例说明

mapper 与 reducer 代码位于 src/microbook/frequency/BuyingFrequency-Analyzer.java。

该图展示了两个 MapReduce job 的执行。如下代码清单展示了

第 1 个 job 的 map 与 reduce 函数：




```
public void map(Object key, Text value, Context context
                ) throwsIOException, InterruptedException {
    List<BuyerRecord> records =
        BuyerRecord.parseAItemLine(value.toString());
    for(BuyerRecord record: records){
        context.write(new Text(record.customerID),
            new IntWritable(record.itemsBrought.size()));
    }
}

public void reduce(Text key, Iterable<IntWritable> values,
                   Context context) {
    int sum = 0;
    for (IntWritableval : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

如图所示，Hadoop 会从输入目录中读取输入文件，并使用第 12 章中所介绍的自定义格式化器读取记录。它会针对每个记录调用 mapper 一次，并将记录作为输入传递进去。

Mapper 会提取出客户 ID 以及客户所购买的商品数量，然后将客户 ID 作为键，将商品数量作为值发射出去。

接下来，Hadoop 会根据键对键值对进行排序，然后针对每个键调用一次 reducer，并将该键的所有值作为输入传递给 reducer。每个 reducer 会计算出每个客户 ID 对应的所有商品数量，然后将客户 ID 作为键，将商品总数作为值发射出去。

然后，第 2 个 job 对结果进行排序。它会读取第 1 个 job 的输出，并将每一行作为参数传递给 map 函数。map 函数会从每一行中提取出客户 ID 和商品数量，然后将商品数量作为键，将客户 ID 作为值发射出去。Hadoop 会根据键对键值对进行排序，这样就会根据商品数量对其进行排序，然后以相同的顺序针对每个键调用一次 reducer。因此，reducer 会按照相同的顺序将其打印出来，从本质上对数据集进行了排序。

由于已经生成了结果，下面来看看绘制的图形。可以在 buyfreq.plot 中找到 gnuplot 文件的源代码，如下所示：

```
set terminal png
set output "buyfreq.png"

set title "Frequency Distribution of Items brought by Buyer";
setylabel "Number of Items Brought";
setxlabel "Buyers Sorted by Items count";
set key left top
set log y
set log x

plot "1.data" using 2 title "Frequency" with linespoints
```

前两行定义了输出格式。该示例使用了 png，不过 gnuplot 还支持很多其他终端，如 screen、pdf 及 eps 等。接下来的 4 行定义了坐

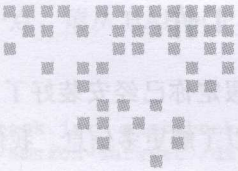
标轴标签与标题，再往下两行定义了每个坐标轴的比例，该图对于两个坐标轴都使用了 log 比例。

最后一行定义了图形。它让 gnuplot 从 1.data 文件中读取数据，然后通过 using 2 使用文件第 2 列中的数据，并且使用直线进行绘制。列之间要通过空白进行分隔。

如果想绘制两列，比如列 1 与列 2 中的数据，那么你应该使用 using 1：2 而非 using 2。

13.4 补充说明

我们可以通过类似的方法计算出大多数分析类型并绘制结果。请参考 Srinath Perera 与 Thilina Gunarathne 编写，Packt 免费发布的《Hadoop MapReduce Cookbook》一书中的部分章节了解更多信息，地址是 http://www.packtpub.com/article/advanced-hadoop-mapreduce-administration。



第 14 章 Chapter 14

关系操作——使用 MapReduce 连接两个数据集（高级）

在 MapReduce 之前，诸如过滤、连接、排序、分组等关系操作是处理大型数据集的主要操作。MapReduce 可以轻松支持过滤、排序等操作。要想了解更多信息，请参考 Anand Rajaraman 与 Jeffrey D.Ullman 所著，剑桥大学出版社 2011 年出版的《Mining of Massive Datasets》一书的 2.3.3 节 Relational-Algebra Operations。

本章将会介绍如何使用 MapReduce 连接两个数据集。它会将购买商品数量最多的 100 个客户与每个客户所购买的商品数据集连接起来，然后生成一个 100 个购买最为频繁的客户所购买的商品列表，并将其作为输出。

14.1 准备工作

1. 本攻略假定你已经安装好了 Hadoop 并将其启动。请参考第 9 章和第 11 章以了解更多信息。我们使用 HADOOP_HOME 来表示 Hadoop 安装目录。

2. 本攻略假定你已经了解了 Hadoop 处理的工作方式。如果不清楚，那么请参考第 10 章以了解更多信息。

3. 下载本章的示例代码以及第 10 章中提及的数据文件。如果可用的计算机不多，那么请从亚马逊数据集中选择一个数据子集。在示例目录中有小数据集。

4. 该示例会用到前面几个攻略中所创建的数据。如果还没有运行，那么请先运行。

14.2 操作步骤

1. 如果之前没有做过，那么请通过如下命令将亚马逊数据集上传到 HDFS 文件系统中：

```
> bin/hadoop dfs -mkdir /data/
> bin/hadoop dfs -mkdir /data/amazon-dataset
> bin/hadoop dfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-dataset/
```

2. 将 hadoop-microbook.jar 文件从 SAMPLE_DIR 复制到 HADOOP_HOME 中。

3. 运行如下 MapReduce job 来创建数据集，该数据集提供了客

户所购买的商品。为了做到这一点，请从 HADOOP_HOME 中运行如下命令：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.join.
Customer2ItemCalculater /data/amazon-dataset /data/join-output1
```

4. 将 MapReduce job 的输出以及上一个攻略的输出复制到输入目录中。注意，文件的名字必须为 mostFrequentBuyers.data 与 itemsByCustomer.data。

```
>bin/hadoop dfs -mkdir /data/join-input
> bin/hadoop dfs -cp /data/join-output1/part-r-00000 /data/join-
input/itemsByCustomer.data
> bin/hadoop dfs -cp /data/frequency-output1/part-r-00000 /data/
join-input/mostFrequentBuyers.data
```

5. 运行第 2 个 MapReduce job。为了做到这一点，请从 HADOOP_HOME 中运行如下命令：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.join.
BuyerRecordJoinJob /data/join-input /data/join-output2
```

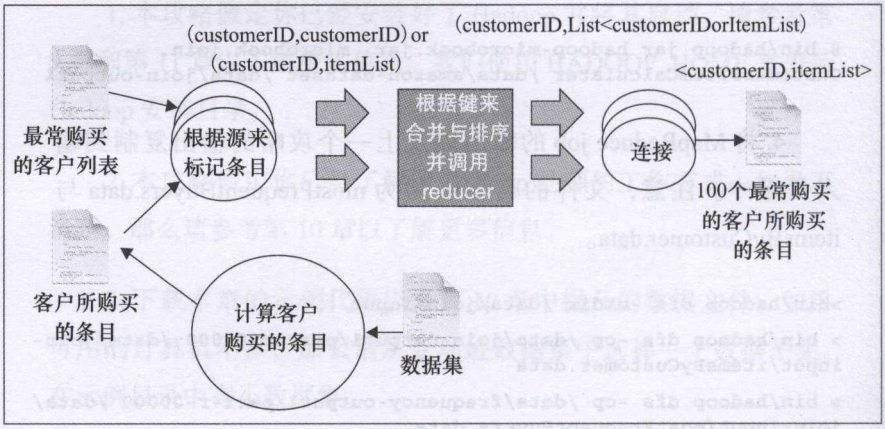
6. 结果位于输出目录 /data/join-output2 中。

14.3　示例说明

Mapper 与 reducer 代码位于 src/microbook/join/BuyerRecordJoinJob.java 中。

第 1 个 MapReduce job 发射出了给定客户 ID 所对应的条目。Mapper 将客户 ID 作为键，将条目 IDs 作为值发射出去。Reducer 接收客户 IDs 作为键，将该客户 ID 所对应的条目 IDs 作为值。它没有

做任何修改将键与值发射出去。



我们通过客户 IDs 将两个数据集连接起来，将两个集合对应的文件放到同一个输入目录中。Hadoop 会从输入目录中读取输入文件，并从该文件中读取记录。它会将传递进来的记录作为输入，并且针对每个记录调用 mapper 一次。

当 mapper 接收到输入时，我们通过 Hadoop 上下文中的 InputSplit 获取到文件名，从而获悉哪一行属于哪一个文件。对于频繁购买的客户列表来说，我们同时将客户 ID 作为键与值发射出去；对于其他数据集来说，我们将客户 ID 作为键，将条目列表作为值发射出去。

```
public void map(Object key, Text value, Context context){
    String currentFile =  ((FileSplit)context
        .getInputSplit()).getPath().getName();
    Matcher matcher = parsingPattern
        .matcher(value.toString());
    if (matcher.find()) {
        String propName = matcher.group(1);
```

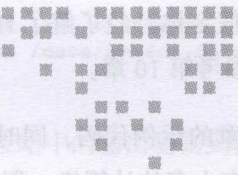
```
        String propValue = matcher.group(2);
        if(currentFile.contains("itemsByCustomer.data")){
            context.write(new Text(propName),
                new Text(propValue));
        }else
                if(currentFile.equals("mostFrequentBuyers.data")){
    context.write(new Text(propName),
                    new Text(propValue));
         }else{
             throw new IOException("Unexpected file "
                 + currentFile);
        }
    }
}
```

Hadoop 会根据键对键值对进行排序，并针对每个唯一的键调用 reducer 一次，将值列表作为第 2 个参数传递进来。reducer 会检查值列表，如果值也包含客户 ID，那么它会将客户 ID 作为键，将条目列表作为值发射出去。

```
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
        InterruptedException {
    boolean isPresent = false;
    String itemList = null;
    for (Text val : values) {
        if(val.toString().equals(key.toString())){
                isPresent = true;
        }else{
                itemList = val.toString();
            }
        }
        if(isPresent && itemList != null){
            context.write(key, new Text(itemList));
        }
    }
```

14.4 补充说明

本攻略的主要想法是将需要连接的信息在 mapper 阶段使用相同的键发送给相同的 reducer，并在 reducer 阶段执行连接逻辑。这种做法也可以用于连接任意类型的数据集。



第 15 章 Chapter 15

使用 MapReduce 实现集合操作（中级）

集合操作对于数据集的处理来说是个非常有用的工具。本章会介绍如何通过 MapReduce 在大规模数据集上执行集合操作。如下的 MapReduce job 会计算出购买了亚马逊销售等级为 100 以下的商品的顾客与之前的攻略中计算出的购买频率最高的顾客之间的差集。

15.1 准备工作

1. 本攻略假定你已经安装好了 Hadoop 并且已经启动。请参考第 9 章和第 11 章以了解更多信息。我们使用 HADOOP_HOME 指代 Hadoop 安装目录。

2. 本攻略假定你已经了解了Hadoop处理的工作方式。如果不清楚，那么请参考第10章。

3. 下载本章的示例代码，同时请下载第10章中所介绍的数据文件。如果没有太多的计算机，那么请从亚马逊数据集中选择一个数据子集。可以在示例目录中找到较小的数据集。

4. 该示例使用了之前攻略中所创建的数据。如果之前尚未运行，那么请先运行。

15.2 操作步骤

1. 如果之前没有做过，那么请通过如下命令将亚马逊数据集上传到HDFS文件系统中：

```
> bin/hadoop dfs -mkdir /data/
> bin/hadoop dfs -mkdir /data/amazon-dataset
> bin/hadoop dfs -put <SMAPLE_DIR>/amazon-meta.txt /data/amazon-
dataset
> bin/Hadoop dfs -mkdir /data/set-input
```

2. 将之前攻略中的输出复制到输出目录中。

```
>bin/hadoop dfs -cp
    /data/frequency-output1/part-r-00000
    /data/set-input/mostFrequentBuyers.data
```

3. 将hadoop-microbook.jar文件从SAMPLE_DIR复制到HADOOP_HOME中。

4. 运行第1个MapReduce job，请从HADOOP_HOME中运行如下命令：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.set.
FindCustomersBroughtFirst100Items /data/amazon-dataset /data/set-
output1
```

5. 将 MapReduce job 与之前攻略中的输出复制到输入目录中：

```
> bin/hadoop dfs -cp /data/set-output1/part-r-00000  /data/set-
input/first100ItemBuyers.data
```

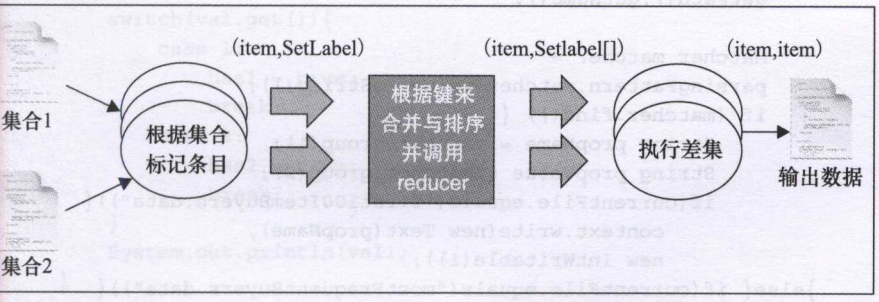
6. 运行第 2 个 MapReduce job，请从 HADOOP_HOME 中运行如下命令：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.set.
BuyersSetDifference /data/set-input /data/set-output2
```

7. 结果位于输出目录 /data/set-output2 中。

15.3　示例说明

Mapper 与 Reducer 代码位于 src/microbook/BuyersSetDifference.java 中。



我们将两个集合 S1 与 S2 之间的差集定义为 S1-S2，表示位于集合 S1 却不在 S2 中的条目。

为了执行差集运算，我们在 mapper 中标识出每个元素来自于哪个集合。接下来，将搜索发送给 reducer，后者只会在条目位于第 1 个集合且不在第 2 个集合中时才将其发射出去。上面的图展示了该 MapReduce job 的执行。如下代码清单展示了 map 与 reduce 函数。

下面来详细看看执行过程。

我们将两个集合的文件放到同一个输入目录中。Hadoop 会从输入目录中读取输入文件，然后读取每个文件的记录。它会将记录作为输入，并且针对每个记录调用一次 mapper。

当 mapper 接收到输入时，我们通过 Hadoop 上下文中的 InputSplit 获取到文件名，从而获悉哪一行属于哪个集合。接下来，我们将集合中的元素作为键，将集合名（1 或是 2）作为值发射出去。

```
public void map(Object key, Text value, Context context) {
    String currentFile =  ((FileSplit)context.getInputSplit()).
    getPath().getName();

    Matcher matcher =
    parsingPattern.matcher(value.toString());
    if (matcher.find()) {
        String propName = matcher.group(1);
        String propValue = matcher.group(2);
        if(currentFile.equals("first100ItemBuyers.data")){
            context.write(new Text(propName),
            new IntWritable(1));
}else{ if(currentFile.equals("mostFrequentBuyers.data")){
            int count = Integer.parseInt(propValue);
            if(count > 100){
                context.write(new Text(propName),
new IntWritable(2));
```

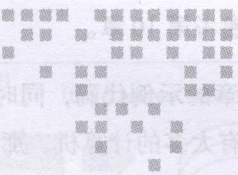
```
            }
        }else{
            throw new IOException("Unexpected file "
+ currentFile);
        }
    } else {
        System.out.println(currentFile
+ ":Unprocessed Line " + value);
    }
}
```

Hadoop 会根据键对键值对进行排序，然后针对每个唯一的键调用 reducer 一次，并将值的列表作为第 2 个参数传递进去。reducer 会检查值的列表，它包含了值所来自的集合名，如果给定的值位于第 1 个集合中且不在第 2 个集合中时，它会将键发射出去。

```
public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException,
        InterruptedException {
    boolean has1 = false;
    boolean has2 = false;
    System.out.print(key + "=");
    for (IntWritable val : values) {
        switch(val.get()){
            case 1:
                has1 = true;
                break;
            case 2:
                has2 = true;
                break;
        }
        System.out.println(val);
    }
    if(has1 && !has2){
        context.write(key, new IntWritable(1));
    }
}
```

15.4 补充说明

我们可以通过 MapReduce 实现大多数集合操作，方式是使用类似的方法将元素所来自的集合标识出来，然后修改 reducer 逻辑，只将相关元素发射出去。比如，我们可以修改 reducer，只将在两个集合中都有值的元素发射出去，从而实现交集。



第 16 章 Chapter 16

使用 MapReduce 实现交叉相关（中级）

交叉相关会检查两件事情同时发生的次数。比如，在亚马逊数据集中，如果两个购买者购买了相同的商品，那么我们就说他们是交叉相关的。通过交叉相关，我们可以计算出两个购买者购买相同商品的数量。

16.1 准备工作

1. 本攻略假定你已经安装好了 Hadoop 并且已经启动。请参考第 9 章和第 11 章以了解更多信息。我们使用 HADOOP_HOME 指代 Hadoop 安装目录。

2. 本攻略假定你已经了解了 Hadoop 处理的工作方式。如果不

清楚，那么请参考第 10 章。

3. 下载本章的示例代码，同时请下载第 10 章中所介绍的数据文件。如果没有太多的计算机，那么请从亚马逊数据集中选择一个数据子集。可以在示例目录中找到较小的数据集。

16.2 操作步骤

1. 如果之前没有做过，那么请从 HADOOP_HOME 中通过如下命令将亚马逊数据集上传到 HDFS 文件系统中：

```
> bin/hadoop dfs -mkdir /data/
> bin/hadoop dfs -mkdir /data/amazon-dataset
> bin/hadoop dfs -put <DATA_DIR>/amazon-meta.txt /data/amazon-dataset/
> bin/hadoop dfs -ls /data/amazon-dataset
```

2. 将 hadoop-microbook.jar 文件从 SAMPLE_DIR 复制到 HADOOP_HOME 中。

3. 从 HADOOP_HOME 中通过如下命令运行 MapReduce job 计算出购买频度：

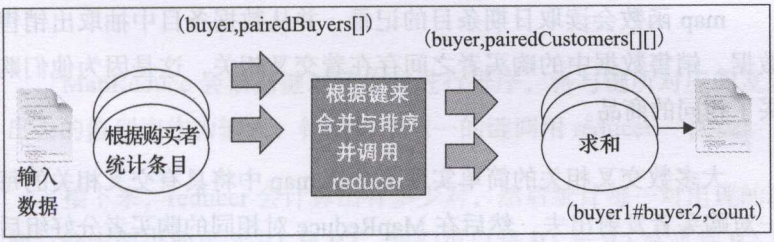
```
$ bin/hadoop jar hadoop-microbook.jar  microbook.crosscorrelation.
CustomerCorrleationFinder /data/amazon-dataset /data/cor-output1
```

4. 结果位于输出目录 /data/cor-output1 中。

16.3 示例说明

mapper 与 reducer 代码位于 src/microbook/Crosscorrelation/Customer-

CorrleationFinder.java 中。



上图展示了 MapReduce job 的执行。此外，如下代码清单展示了第 1 个 job 的 map 与 reduce 函数：

```
public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
    List<BuyerRecord> records =
        BuyerRecord.parseAItemLine(value.toString());
    List<String> customers = new ArrayList<String>();

    for(BuyerRecord record: records){
        customers.add(record.customerID);
    }

    for(int i =0;i< records.size();i++){
        StringBuffer buf = new StringBuffer();
        int index = 0;
        for(String customer:customers){
            if(index != i){
                buf.append(customer).append(",");
            }
        }
        context.write(new Text(customers.get(i)),
        new Text(buf.toString()));
    }
}
```

如上图所示，Hadoop 会从输入目录中读取输入文件，并使用第 12 章中所介绍的自定义格式化器读取记录。它会将记录作为输

入，并针对每个记录调用一次 mapper。

map 函数会读取日期条目的记录，并从数据条目中抽取出销售数据。销售数据中的购买者之间存在着交叉相关，这是因为他们购买了相同的商品。

大多数交叉相关的简单实现都会从 map 中将具有交叉相关的每一对购买者发射出去，然后在 MapReduce 对相同的购买者分好组后在 reducer 函数中计算出出现的次数。

不过，这会产生不同购买者数量二次方以上的结果，对于大型数据集来说，这可能是个非常庞大的数字。因此，我们将使用更加优化的版本，限制键的数量。

mapper 将购买者作为键发射出去，并且将与该 mapper 所匹配的所有其他购买者作为键发射出去。

```
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
        InterruptedException {
    Set<String> customerSet = new HashSet<String>();
    for(Text text: values){
        String[] split = text.toString().split(",");
        for(String token:split){
            customerSet.add(token);
        }
    }
    StringBuffer buf = new StringBuffer();
    for(String customer: customerSet){
        if(customer.compareTo(key.toString()) < 0){
            buf.append(customer).append(",");
        }
    }
    buf.append("|").append(Integer.MAX_VALUE)
```

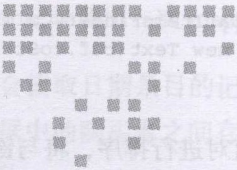
```
        .append("|").append(SimilarItemsFinder.Color.White);
    context.write(new Text(key), new Text(buf.toString()));
}
```

MapReduce 会根据键对键值对进行排序，将与键所对应的发射出去的值列表作为输入，针对每个唯一的键调用 reducer 一次。

接下来，reducer 会计算出有多少对，然后统计每一对出现的次数。给定两个购买者 B1 与 B2，我们可以将 B1 与 B2 对或是 B2 与 B1 对发射出去，这会生成重复的数据。我们可以只当 B1 在字母表中位于 B2 前时才将其发射出去来避免这一点。

16.4　补充说明

交叉相关是 MapReduce 所面临的难题之一，因为它会生成大量的数据对。通常情况下，我们只用 MapReduce 处理小型数据集。



Chapter 17 第 17 章

使用 MapReduce 实现简单搜索（中级）

根据 Google 所述，文本搜索是 MapReduce 最初几个用例之一，他们将 MapReduce 作为与其搜索平台相关的文本处理编程模型。

一般来说，搜索是通过一个反向索引实现的。反向索引指的是单词到包含该单词的数据条目的映射。给定一个搜索查询，我们会找出包含了查询单词的所有文档。Web 搜索的一个复杂性在于有太多的结果，而我们只需要显示出重要的查询。不过，根据重要程度对文档进行评级则不在本书的讨论范围之内。

本章将会介绍如何通过 MapReduce 构建一个简单的基于反向索引的搜索。

17.1　准备工作

1. 本攻略假定你已经安装好了 Hadoop 并且已经启动。请参考第 9 章与第 11 章以了解更多信息。我们使用 HADOOP_HOME 指代 Hadoop 安装目录。

2. 本攻略假定你已经了解了 Hadoop 处理的工作方式。如果不清楚，那么请参考第 10 章。

3. 下载本章的示例代码，同时请下载第 10 章中所介绍的数据文件。如果没有太多的计算机，那么请从亚马逊数据集中选择一个数据子集。可以在示例目录中找到较小的数据集。

17.2　操作步骤

1. 如果之前没有做过，那么请从 HADOOP_HOME 中通过如下命令将亚马逊数据集上传到 HDFS 文件系统中：

```
> bin/hadoop dfs -mkdir /data/
> bin/hadoop dfs -mkdir /data/amazon-dataset
> bin/hadoop dfs -put <DATA_DIR>/amazon-meta.txt /data/amazon-
dataset/
```

2. 将 hadoop-microbook.jar 文件从 SAMPLE_DIR 复制到 HADOOP_HOME 中。

3. 从 HADOOP_HOME 中通过如下命令运行 MapReduce job 计算出购买频度：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.search.
TitleInvertedIndexGenerator /data/amazon-dataset /data/search-
output
```

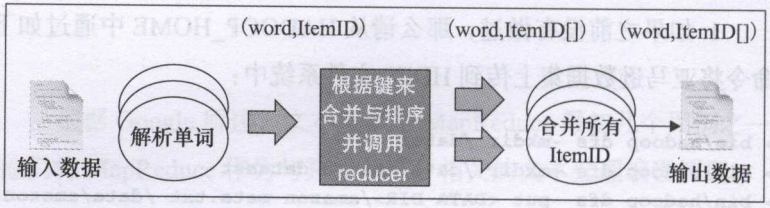
4. 结果位于输出目录 /data/search-output 中。

5. 运行如下命令从服务器下载结果文件，并使用 MapReduce job 所构建的索引搜索单词 Cycling，它应该打印出标题中含有单词 Cycling 的条目。

```
$ bin/hadoop dfs -get /data/search-output/part-r-00000
invetedIndex.data

$ java -cp hadoop-microbook.jar microbook.search.
IndexBasedTitleSearch invetedIndex.data Cycling
```

17.3 示例说明

Mapper 与 reducer 代码位于 src/microbook/search/TitleInvertedIndex-Generator.java 中。



上图展示了两个 MapReduce job 的执行。此外，如下代码清单展示了第 1 个 job 的 map 与 reduce 函数。

如上图所示，Hadoop 会从输入目录中读取输入文件，并使用之前在第 12 章中所介绍的自定义格式化器读取记录。它会将记录作为输入，并针对每个记录调用一次 mapper。

map 函数会从记录中读取条目的标题，对其进行分词，然后将标题中的每个单词作为键，将条目 ID 作为值发射出去。

```
public void map(Object key, Text value, Context context) {
    List<BuyerRecord> records =
BuyerRecord.parseAItemLine(value.toString());
    for (BuyerRecord record : records) {
        for(ItemData item: record.itemsBrought){
            StringTokenizer itr =
new StringTokenizer(item.title);
            while (itr.hasMoreTokens()) {
                String token =
itr.nextToken().replaceAll("[^A-z0-9]", "");
                if(token.length() > 0){
                    context.write(new Text(token),
                    new Text(
                    pad(String.valueOf(item.salesrank))
                    + "#" +item.itemID));
                }
            }
        }
    }
}
```

MapReduce 会根据键对键值对进行排序，将针对于该键所发射出的值列表作为输入，并针对每个唯一的键调用 reducer 一次。

每个 reducer 都会将接收到的单词作为键，将条目 IDs 列表作为值，并且将其原样发射出去。输出则是个反向索引。

```
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
        InterruptedException {
    TreeSet<String> set = new TreeSet<String>();
    for (Text valtemp : values) {
        set.add(valtemp.toString());
    }

    StringBuffer buf = new StringBuffer();
    for (String val : set) {
        buf.append(val).append(",");
    }
```

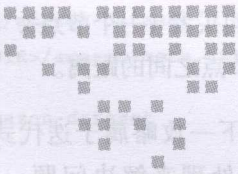
```
    context.write(key, new Text(buf.toString()));
}
```

如下代码清单展示了搜索程序的代码。该搜索程序将反向索引加载到内存中，在搜索某个单词时，它会寻找与该单词相对应的条目 IDs 并将其显示出来。

```
String line = br.readLine();
while (line != null) {
    Matcher matcher = parsingPattern.matcher(line);
    if (matcher.find()) {
        String key = matcher.group(1);
        String value = matcher.group(2);

        String[] tokens = value.split(",");
        invertedIndex.put(key, tokens);
        line = br.readLine();
    }
}

String searchQuery = "Cycling";
String[] tokens = invertedIndex.get(searchQuery);
if (tokens != null) {
    for (String token : tokens) {
        System.out.println(Arrays.toString(token.split("#")));
        System.out.println(token.split("#")[1]);
    }
}
```

17.4 补充说明

我们可以通过索引从大型数据集中快速寻找到数据，也可以利用相同的模式构建索引以支持快速搜索。



第 18 章 Chapter 18

使用 MapReduce 实现简单的图操作（高级）

图是我们经常遇到的另一类数据。图的主要应用场景之一就是社交网络；人们可以通过搜索图来寻找感兴趣的模式。本章将介绍如何通过 MapReduce 执行简单的图操作与图遍历。

本章将会利用第 10 章的结果。每个购买者都是一个节点，如果两个购买者购买了相同的商品，那么这两个节点间就会有一条边。

下面展示了一个示例输入：

```
AR1T36GLLAFFX  A26TSW6AI59ZCV,A39LRCAB9G8F21,ABT9YLRGT4ISP|Gray
```

第 1 个标记是个节点，逗号分隔的值是第 1 个节点与之存在边的节点列表。最后一个值是节点的颜色。该结构将用于图遍历算法。

给定一个购买者（一个节点），本攻略将会遍历图，并计算出给定节点到其他节点之间的距离。

本攻略与下一攻略属于迭代式MapReduce，我们无法通过对数据的一次性处理来解决问题。迭代式MapReduce会通过一个MapReduce job处理数据多次，直到计算出了给定节点到所有其他节点之间的距离为止。

18.1 准备工作

1. 本攻略假定你已经安装好了Hadoop并且已经启动。请参考第9章和第11章以了解更多信息。我们使用HADOOP_HOME指代Hadoop安装目录。

2. 本攻略假定你已经了解了Hadoop处理的工作方式。如果不清楚，那么请参考第10章。

3. 下载本章的示例代码，同时请下载第10章中所介绍的数据文件。如果没有太多的计算机，那么请从亚马逊数据集中选择一个数据子集。可以在示例目录中找到较小的数据集。

18.2 操作步骤

1. 切换到目录HADOOP_HOME，将hadoop-microbook.jar文件从SAMPLE_DIR复制到HADOOP_HOME。

2. 如果之前没有做过，那么请从HADOOP_HOME中通过如下命令将亚马逊数据集上传到HDFS文件系统中：

```
> bin/hadoop dfs -mkdir /data/
```

```
> bin/hadoop dfs -mkdir /data/amazon-dataset
> bin/hadoop dfs -put <DATA_DIR>/amazon-meta.txt /data/amazon-
dataset/
> bin/hadoop dfs -ls /data/amazon-dataset
```

3. 运行如下命令生成图：

```
> bin/hadoop jar hadoop-microbook.jar  microbook.graph.
GraphGenerator /data/amazon-dataset /data/graph-output1
```

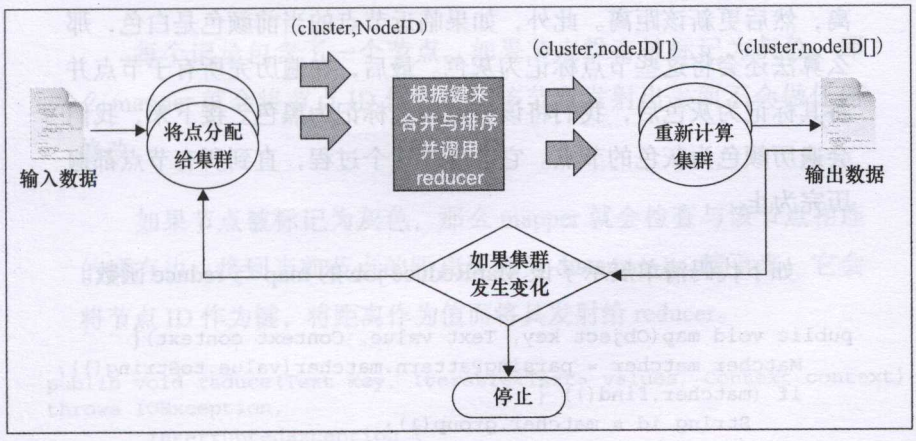
4. 通过如下命令运行 MapReduce job 计算出图距离：

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.graph.
SimilarItemsFinder /data/graph-output1 /data/graph-output2
```

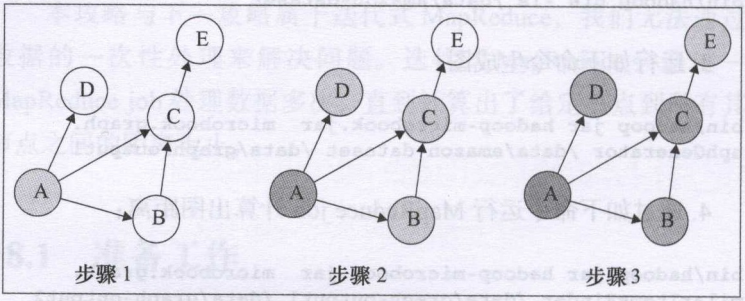
5. 结果位于 /data/graph-output2 中，它包含了所有迭代的结果，你应该查看最后一次迭代的结果。

18.3　示例说明

mapper 与 reducer 代码位于 src/microbook/SimilarItemsFinder.java 中。



上图展示了两个 MapReduce job 的执行与驱动代码。启动代码重复 map reduce job，直到图遍历完成为止。



算法通过为图节点着色进行说明。每个节点一开始都是白色的，除了开始遍历的那个节点，它是灰色的。生成图之后，代码会将该节点标记为灰色。如果需要改变起始节点，那么可以通过编辑图来实现。

如上图所示，在每一步骤中，MapReduce job 会处理标记为灰色的节点，并计算出与通过边连接到灰色节点的其他节点之间的距离，然后更新该距离。此外，如果临近节点的当前颜色是白色，那么算法还会将这些节点标记为灰色。最后，在遍历完所有子节点并将其标记为灰色后，我们将该节点颜色标记为黑色。接下来，我们会遍历颜色为灰色的节点。它会继续这个过程，直到所有节点都遍历完为止。

如下代码清单展示了该 MapReduce job 的 map 与 reduce 函数。

```
public void map(Object key, Text value, Context context){
    Matcher matcher = parsingPattern.matcher(value.toString());
    if (matcher.find()) {
        String id = matcher.group(1);
        String val = matcher.group(2);
```

```
        GNode node = new GNode(id, val);
        if(node.color == Color.Gray){
            node.color = Color.Black;
            context.write(new Text(id),
            new Text(node.toString()));
            for(String e: node.edges){
                GNode nNode = new GNode(e, (String[])null);
                nNode.minDistance = node.minDistance+1;
                nNode.color = Color.Red;
                context.write(new Text(e),
                new Text(nNode.toString()));
            }
        }else{
            context.write(new Text(id), new Text(val));
        }
    } else {
        System.out.println("Unprocessed Line " + value);
    }
}
```

如上图所示，Hadoop 会从输入目录中读取输入文件，并使用之前在第 12 章中所介绍的自定义格式化器读取记录。它会将记录作为输入，并针对每个记录调用一次 mapper。

每个记录包含了一个节点。如果节点没有被标记为灰色，那么 mapper 就会将节点 ID 作为键将该节点发射出去而不会做任何修改。

如果节点被标记为灰色，那么 mapper 就会检查与该节点相连的所有边，将到当前节点的距离更新为 distance+1。接下来，它会将节点 ID 作为键，将距离作为值而将其发射给 reducer。

```
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
        InterruptedException {
    GNode originalNode =  null;
```

```
    boolean hasRedNodes = false;
    int minDistance = Integer.MAX_VALUE;
    for(Text val: values){
        GNode node = new GNode(key.toString(),val.toString());
        if(node.color == Color.Black ||
node.color == Color.White){
            originalNode = node;
        }else if(node.color == Color.Red){
            hasRedNodes = true;
        }
        if(minDistance > node.minDistance){
            minDistance = node.minDistance;
        }
    }
    if(originalNode != null){
        originalNode.minDistance = minDistance;
        if(originalNode.color == Color.White && hasRedNodes){
            originalNode.color = Color.Gray;
        }
        context.write(key, new Text(originalNode.toString()));
    }
}
```

MapReduce会根据键对键值对进行排序，将针对于该键所发射出的值列表作为输入，并针对每个唯一的键调用reducer一次。

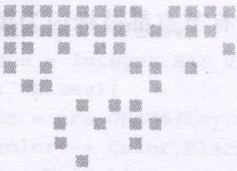
每个reducer都会接收到关于节点与距离的键值对信息，这是在遇到节点时由mapper计算出来的。如果距离更新小于当前的节点距离，那么reducer就会更新节点中的距离。接下来，它会将节点ID作为键，将节点信息作为值发射出去。

驱动会重复该过程，直到所有节点都被标记为黑色并且距离得到更新为止。一开始，只有一个节点被标记为灰色，其他所有节点都是白色。在每次执行时，MapReduce job会将连接到第1个节点的其他节点标记为灰色并更新距离。它会将遍历过的节点标记为黑色。

我们继续这个过程，直到所有节点都被标记为黑色并更新了距离为止。

18.4　补充说明

用户可以通过本攻略所介绍的迭代式、基于 MapReduce 的解决方案处理很多图算法，如图搜索等。



Chapter 19 第 19 章

使用 MapReduce 实现 Kmeans（高级）

如果想要从大规模数据集中找到或是计算出感兴趣的信息，通常情况下，我们要计算的算法要比这里讨论的更为复杂。这样的算法有很多（比如，聚类、协同过滤以及数据挖掘算法等）。本章将会实现一个名为 Kmeans 的算法，它是聚类算法的一种。

假定亚马逊数据集中包含了客户的位置。由于该信息并不存在，因此我们通过从 IP 地址到经纬度转换的数据集（http://www.infochimps.com/datasets/united-states-ip-address-to-geolocation-data）随机获取一些随机值来创建数据集。

如果可以根据地理位置对客户进行分组，那么就可以提供更加

专业且本地化的服务。本章将会通过 MapReduce 实现 Kmeans 聚类算法，并使用它确定基于客户地理位置的聚类。

聚类算法会将一个数据集划分为几个叫作聚类的组，这样相同聚类中的数据点就要比不同聚类中的更加接近一些。在该示例中，我们使用数据点中心来表示聚类。

19.1　准备工作

1. 本攻略假定你已经安装好了 Hadoop 并且已经启动。请参考第 9 章和第 11 章以了解更多信息。我们使用 HADOOP_HOME 指代 Hadoop 安装目录。

2. 本攻略假定你已经了解了 Hadoop 处理的工作方式。如果不清楚，那么请参考第 10 章。

3. 下载本章的示例代码，同时请从 http://www.infochimps.com/datasets/united-states-ip-address-togeolocation-data 下载数据文件。

19.2　操作步骤

1. 将 geo-location 数据集解压缩到一个目录中，我们称该目录为 GEO_DATA_DIR。

2. 切换到 HADOOP_HOME 目录，将 hadoop-microbook.jar 文件从 SAMPLE_DIR 复制到 HADOOP_HOME。

3. 运行如下命令生成示例数据集与初始化聚类，这会生成一个名为 customer-geo.data 的文件。

```
> java -cp hadoop-microbook.jar microbook.kmean.GenerateGeoDataset
GEO_DATA_DIR/ip_blocks_us_geo.tsv customer-geo.data
```

4. 将数据集上传到 HDFS 文件系统中。

```
> bin/hadoop dfs -mkdir /data/
> bin/hadoop dfs -mkdir /data/kmeans/
> bin/hadoop dfs -mkdir /data/kmeans-input/
> bin/hadoop dfs -put HADOOP_HOME/customer-geo.data /data/kmeans-
input/
```

5. 从 HADOOP_HOME 中执行如下命令来运行 MapReduce job 计算聚类。这里的 5 表示迭代次数，10 表示聚类数量。

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.kmean.
KmeanCluster /data/kmeans-input/ /data/kmeans-output 5 10
```

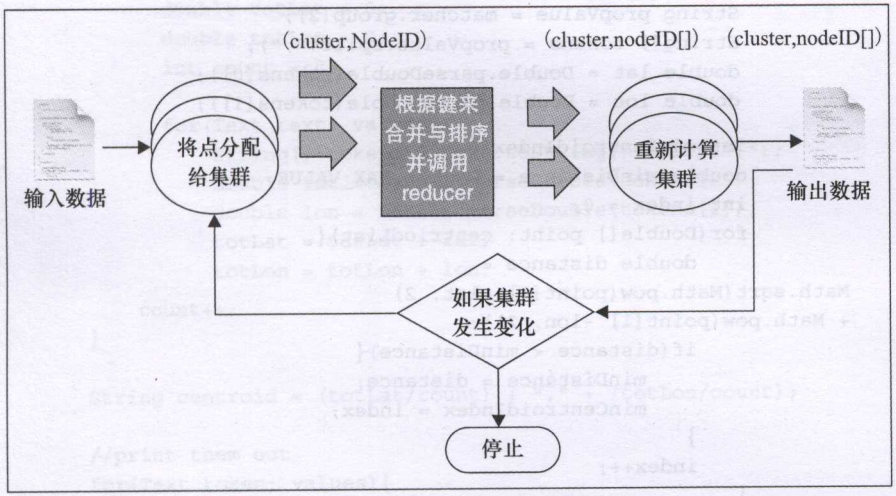
6. 执行完毕后会将最终的聚类打印到控制台，同时结果也会输出到目录 /data/kmeans-output 中。

19.3 示例说明

mapper 与 reducer 代码位于 src/microbook/KmeanCluster.java。该类包含了 map 与 reduce 函数以及驱动程序。

开始后，驱动程序会生成 10 个随机聚类，并将其写到 HDFS 文件系统中的文件中去。接下来，它会在每次迭代时调用一次 MapReduce job。

下图展示了两个 MapReduce job 的执行。本攻略使用了迭代式的 MapReduce，我们会迭代运行 MapReduce 程序，直到结果收敛为止。



当 MapReduce job 被调用后，Hadoop 会调用 mapper 类的设置方法，mapper 会从 HDFS 文件系统中读取当前聚类并将其加载到内存中。

如上图所示，Hadoop 会从输入目录中读取输入文件，并使用之前在第 12 章中所介绍的自定义格式化器读取记录。它会将记录作为输入，并针对每个记录调用一次 mapper。

当调用了 mapper 后，它会解析输入并抽取出位置信息，找到与当前位置最为接近的聚类，并将聚类 ID 作为键，将位置作为值发射出去。如下代码清单展示了 map 函数：

```
public void map(Object key, Text value, Context context) {
    Matcher matcher = parsingPattern.matcher(value.toString());
    if (matcher.find()) {
        String propName = matcher.group(1);
```

```
        String propValue = matcher.group(2);
        String[] tokens = propValue.split(",");
        double lat = Double.parseDouble(tokens[0]);
        double lon = Double.parseDouble(tokens[1]);

        int minCentroidIndex = -1;
        double minDistance = Double.MAX_VALUE;
        int index = 0;
        for(Double[] point: centriodList){
            double distance =
Math.sqrt(Math.pow(point[0] -lat, 2)
+ Math.pow(point[1] -lon, 2));
            if(distance < minDistance){
                minDistance = distance;
                minCentroidIndex = index;
            }
            index++;
        }

        Double[] centriod = centriodList.get(minCentroidIndex);
        String centriodAsStr = centriod[0] + "," + centriod[1];
        String point = lat +"," + lon;
        context.write(new Text(centriodAsStr), new Text(point));
    }
}
```

MapReduce会根据键对键值对进行排序，将针对于该键所发射出的值列表作为输入，并针对每个唯一的键调用reducer一次。

reducer会将接收到的聚类ID作为键，将与该聚类ID对应的发射出去的位置列表作为值。这样，reducer就可以重新计算聚类中所有位置的平均值，然后通过该聚类信息更新HDFS位置。如下代码清单展示了reducer函数：

```
public void reduce(Text key, Iterable<Text> values,
Context context)
{
    context.write(key, key);
    //recalcualte clusters
```

```
        double totLat = 0;
        double totLon = 0;
        int count = 0;

        for(Text text: values){
            String[] tokens = text.toString().split(",");
            double lat = Double.parseDouble(tokens[0]);
            double lon = Double.parseDouble(tokens[1]);
            totLat = totLat + lat;
            totLon = totLon + lon;

        count++;
    }

    String centroid = (totLat/count) + "," + (totLon/count);

    //print them out
    for(Text token: values){
        context.write(new Text(token), new Text(centroid));

    }
    FileSystem fs =FileSystem.get(context.getConfiguration());

    BufferedWriter bw = new BufferedWriter(
new OutputStreamWriter(fs.create(new Path("/data/kmeans/clusters.
data"), true)));
    bw.write(centroid);bw.write("\n");
    bw.close();

}
```

驱动程序会在每次迭代时重复上述步骤，直到给定 MapReduce job 的输入聚类与输出聚类相同为止。

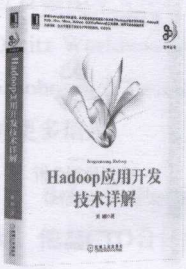
算法从随机聚类点开始。在每一步中，它会将位置分配给聚类点，在 reduce 阶段，它会调整每个聚类点，使之等于分配给每个聚类的位置的平均值。在每次迭代中，聚类会移动，直到聚类最适合该数据集为止。当聚类在迭代中不再变化时我们就停下来。

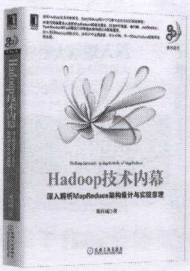
19.4 补充说明

Kmeans算法的一个限制在于我们需要知道数据集中聚类的数量，还有很多其他的聚类算法。你可以从Anand Rajaraman与Jeffrey D.Ullman所著的，剑桥大学出版社2011年出版的免费图书《Mining of Massive Datasets》的第7章中了解到关于这些算法的更多信息。

推荐阅读


Hadoop实战


Hadoop应用开发
技术详解


Hadoop技术内幕
深入解析MapReduce架构设计与实现原理


Hadoop技术内幕
深入解析YARN架构设计与实现原理


HBase企业应用
开发实战


Mahout算法解析
与案例实战


Spark大数据处理
技术、应用与性能优化


Storm实时数据处理

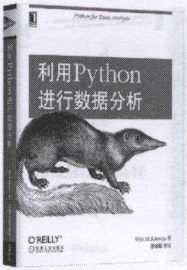

Splunk大数据分析

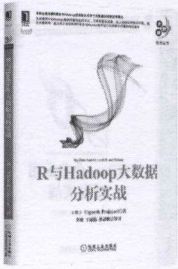
推荐阅读


深入理解大数据
大数据处理与编程实践
UNDERSTANDING BIG DATA
BIG DATA

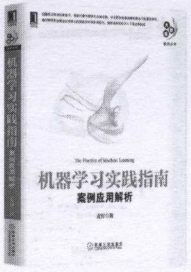

实用数据分析


Clojure数据分析秘笈


利用Python
进行数据分析
O'REILLY


R与Hadoop大数据
分析实战


Hadoop集群与安全


机器学习实践指南
案例应用解析


大规模分布式系统
架构与设计实战


大数据管理
数据集成的技术、方法与
最佳实践

作者简介

Steve Hoffman 具有30年的软件开发经验，拥有伊利诺伊大学香槟分校计算机工程学士学位以及德保罗大学计算机科学硕士学位。他目前是Orbitz Worldwide的首席工程师。通过http://bit.ly/bacoboy或者Twitter @bacoboy可以了解关于Steve的更多信息。

Srinath Perera WSO2公司的高级软件架构师，他与CTO合作负责WSO2平台的整体架构。此外，他还是Lanka软件基金会的研究科学家，并且担任莫勒图沃大学计算机科学与工程学院的客座讲师。他是Apache Axis2开源项目的联合创始人，从2002年开始就参与Apache Web Service项目，是Apache软件基金会的成员以及Apache Web Service项目的PMC。他也是Apache开源的Axis、Axis2以及Geronimo项目的提交者。

译者简介

张龙 InfoQ中文站编辑，9年技术开发、管理与培训经验，多本技术图书译者。喜好用简单技术解决复杂问题，跑步与足球爱好者，目前从事法律互联网相关工作，希望能用互联网思维与技术改变法律这一传统行业。

封面设计 · 王建敏

通过阅读本书，你将学到：

- 理解Flume架构
- 从Apache下载并安装开源的Flume
- 探索何时使用内存通道，何时使用基于文件的通道
- 理解并配置Hadoop文件系统（HDFS）接收器
- 学习如何使用接收器组来创建冗余的数据流
- 配置并使用各种源来提取数据
- 监测数据记录，根据负载内容路由到不同目的地或者多个目的地
- 即时将数据转换到Hadoop
- 监控数据流
- 编写并运行一个简单的MapReduce程序
- 理解Hadoop的工作原理以及如何编写自定义的格式化器
- 使用Hadoop实现分析、正交相关以及集合操作
- 编写简单的Hadoop程序执行搜索
- 通过编写Hadoop程序实现数据的连接
- 实现图操作与集群

[PACKT] PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/大数据

ISBN 978-7-111-50207-4

9 787111 502074 >

定价：39.00元